Process Fragmentation, Distribution and Execution using an Event-Based Interaction Scheme

Pieter Hens[a,*], Monique Snoeck[a], Geert Poels[b], Manu De Backer[a,b,c]

[a] *Department of Decision Sciences and Information Management, Katholieke Universiteit Leuven, Naamsestraat 69, B-3000 Leuven, Belgium*
[b] *Department of Management Information and Operations Management, Universiteit Gent, Tweekerkenstraat 2, B-9000 Ghent, Belgium*
[c] *Department of Management Information Systems, Universiteit Antwerpen, Prinsstraat 13, B-2000 Antwerp, Belgium*

Abstract

The combination of Service Oriented Architectures and Business Processes creates an enactment environment in which processes can be deployed and executed automatically. From a managerial and technical point of view, the interpretation, control and execution of a process flow happens very often at one point in the organizational and IT structure. This creates an inflexible environment in which control over and visibility of cross-departmental processes cannot be distributed across these organizational entities. Although the process model may need to be designed as a whole (to have an end-to-end definition), the actual execution of the process may need to be distributed across all participating partners. There are several ways to achieve this distribution. In this paper we look at an event-based process deployment and execution infrastructure in which a process model can be automatically partitioned and distributed over different enactment entities, provided some given distribution definition. We compare the performance and flexibility of the proposed technique with other approaches and discuss the potential advantages and drawbacks of the event-based distribution.

Keywords: Business Process Enactment / Execution, Event Based Architecture, Distributed Business Processes, Service Oriented Architecture, Workflow Management

1. Introduction

Process-aware information systems (PAISs) are becoming more and more integrated into todays business environments (Dumas et al., 2005). In combination with Service Oriented Architectures (SOA) it becomes possible to automatically execute a business

*Corresponding Author; postal address: Naamsestraat 69, 3000 Leuven, Belgium; phone: +3216326886, fax: +3216326624

Email addresses: pieter.hens@econ.kuleuven.be (Pieter Hens), monique.snoeck@econ.kuleuven.be (Monique Snoeck), geert.poels@ugent.be (Geert Poels), manu.debacker@econ.kuleuven.be (Manu De Backer)

November 16, 2013

process by a process engine. Executing a business process means coordinating the described work, invoking the correct services, sending messages to other business processes, adding tasks to the inbox of task managers, and choosing the correct process flow paths (Hollingsworth, 1995). The combination of SOA and process engines achieves a separation between the logic and functionality of a business process, where the functionality is contained in space- and logically- distributed (IT- or human-) services and the process logic is contained in the process model executed by the process engine.

In current process execution architectures, each process engine is responsible for the execution of one or more business processes, which means that the logic of a business process is logically and spatially located at one point in the enterprise and IT architecture. This concept is known as *centralized process execution* (even though the execution of the tasks/functionality may be distributed). However, it is not uncommon that processes are cross-departmental or even cross-organizational, whereby it is not viable that one single entity has full control over the entire process flow, or even has visibility of the entire process flow. Using a single process engine to operate the complete process flow also means that the responsibility of the process execution lies with one organizational entity. For example, a specific process part can be an important IP asset of a specific company, making it not desirable to hand over the coordination of these process parts to other partners. The process model may need to be designed as a whole (to have an end-to-end definition), but the actual execution of the process may need to be distributed across all participating partners. This is the case when process parts are for example being offshored or outsourced. Like the physical and logical distribution of services, it could be possible that the business process logic should also be distributed across organizational entities so as to allow the distribution of the process control, visibility and responsibility of the process execution (Khalaf and Leymann, 2006; Fdhila et al., 2009).

A way to achieve this distribution is *business process model fragmentation*. Business process model fragmentation is the process of splitting a process model that was modeled as a whole into logically different, smaller model fragments with the intention to distribute the fragments over different execution and controlling partners (Khalaf et al., 2008). Some research has already been devoted to the fragmentation and distribution of the process logic. For example, Chafle et al. (2004) and Zhai et al. (2007) define automatic derivation rules to transform a process into multiple (smaller) process parts. The resulting distributed fragments trigger each other in a directed, request based style creating a tight coupling between the fragments. In contrast, Jennings et al. (2000) and Li et al. (2010) provide an infrastructure in which the distributed fragments become autonomous and self-contained. Both approaches do however not provide automatic derivation rules for the fragments and only support basic workflow patterns. Most approaches also do not focus on the volatility of a process model. To cope with the changing business environment, a process model deployment should enable frequent and quick changes. In a distributed deployment however, changing the model can be costly as each (physically) distributed node needs to be changed independently. For a more detailed discussion on existing literature we refer to Section 8.

1.1. Research Objectives

In this paper we aim to investigate the pros and cons of a process distribution approach that uses an *event-based* interaction scheme. As mentioned in the previous section, the distribution of the process execution facilitates the distribution of control and

visibility. However, fragmentation and distribution of a business process can increase the coordination overhead of the process execution, hereby decreasing the performance. Moreover, since the process logic is scattered in the enterprise architecture, there is a potential increase in the complexity of process changes.

To be able to investigate the possible advantages and disadvantages of process distribution, we propose an event-based architecture, which removes as many hurdles as possible that may negatively impact the adoption of process model fragmentation and distributed process execution. In particular, we propose an approach that pays special attention to a fully automated way of deploying processes and to minimizing the cost of process model changes in the a distributed infrastructure. Its starting point is a fully specified process model including a definition of the desired process model distribution (e.g. distribute the model according to the owner of each activity in the process) which needs to be deployed in the IT and organizational architecture. The process flow (process logic) will become distributed in the architecture, mimicking the distributed behavior of the enterprise itself. Process fragments can be spread out among different organizational and technical entities (e.g. placing a fragment close to the data or service it operates on), whereby each fragment is executed by a lightweight, autonomous process engine.

We specifically look for a distributed approach that satisfies the following criteria:

REQ1 A *non-intrusive approach*, where, provided a fully specified process model and a definition of the desired fragments, the executable process fragments are created automatically (without interference of the original process modeler). This ensures that the process modeler does not have to know the technical details of process distribution and of the runtime architecture. A mechanism should therefore be provided that takes a modeled process flow and the definition of the process fragments as input and outputs different, executable process parts. The definition of the process fragments consists of the *grouping of activities* from the original process model that form together one distributed process part (Fdhila et al., 2009). This grouping can be defined manually or automatically according to some distribution strategy, e.g. minimizing network traffic (Chafle et al., 2004).

REQ2 After fragmentation, the created executable process fragments are each executed by a *dedicated*, *lightweight* process engine. To accommodate for the visibility and security arguments of process model fragmentation, each engine should run a different part of the original process flow, where the intersection between two engines is zero. This ensures that process parts can be distributed to different controlling partners, with each partner only having control and visibility of their own process parts (Khalaf and Leymann, 2006).

REQ3 The process runtime architecture should be fault-tolerant (*robust*) and should keep coordination cost low as not to compromise performance (*scalable*).

REQ4 The environment should be *flexible* enough to allow process model changes and changes to the deployment structure during runtime. Moreover, since we require the system to be robust and scalable (see REQ3), deploying a modified process model should have a *limited impact* on the enactment environment. Lowering the impact of changes on the runtime architecture increases the availability of the process execution system and therefore increases the level of operational performance (Gray and Siewiorek, 1991; Mühl et al., 2006).

In this paper we focus on REQ1-3. REQ4 is only briefly discussed in section 6. A more elaborate discussion of REQ4 can be found in (Hens et al., 2013)(Hens et al., 2014). After describing the approach, we compare its performance and flexibility with other approaches and discuss the potential advantages and drawbacks of event-based distribution.

1.2. Research Situation

Figure 1 demonstrates the starting point of this research. This research focuses on *executable business process models*, i.e. a process model designed in some executable language. Before an executable business process model is created, the goals it should achieve and a high-level process model are designed. Next, all technical specifications are added which allow the model to be deployed in the execution architecture.

When a process model is deployed in a distributed coordination architecture, the distribution groupings need to be defined in the model. The distribution grouping defines the activities in the process model that logically belong together, i.e. the activities that are deployed on the same node in the distributed environment. A specific distribution criterion can be used to define these groupings (e.g. group all activities that invoke the same service or are performed by the same human performer). The definition can be done manually or automatically.

After the specification of the process model and distribution groupings, the process model is fragmented into different parts. The process model is split according to the groupings defined in the previous step. These split models can hereafter be deployed on their designated process engines in the distributed environment. After deployment, the process model can be enacted.

This research starts from the fragmentation step. Its starting point is therefore a fully specified, executable process model, including a definition of the desired process model distribution. This research proposes an approach where the fragmentation can be automatically performed and describes an enactment environment to execute the split process model. Moreover, how changes can be applied to the running architecture is also briefly discussed. These features are hereafter compared to standard process execution architectures and other process distribution architectures as to discuss the advantages and disadvantages of process distribution using an event-based fragmentation.

The core of our approach consists of three parts. First, a formal transformation to *automatically* transform a business process model into distributed autonomous fragments is given (REQ1). Note that to accomplish this fragmentation, we rewrite the original process model using event execution semantics (see section 3). This rewriting is only done as an enabler for the process distribution. The formal approach enables the adaptation of the transformation to any process modeling and execution language (Section 4). Second, the design of a proof of concept execution architecture and execution semantics that describe how to interpret and execute the distributed process flow in a flexible manner is given (REQ2). An example implementation of the transformation and execution architecture using BPMN2.0 as the process modeling language is provided (Section 5). REQ4 states that the environment should also support process change. Given the complexity of this topic, this is handled in a separate paper. However, to make this paper sufficiently self-contained a third section (Section 6) provides a brief overview of how process changes can be performed in the distributed environment (REQ4).

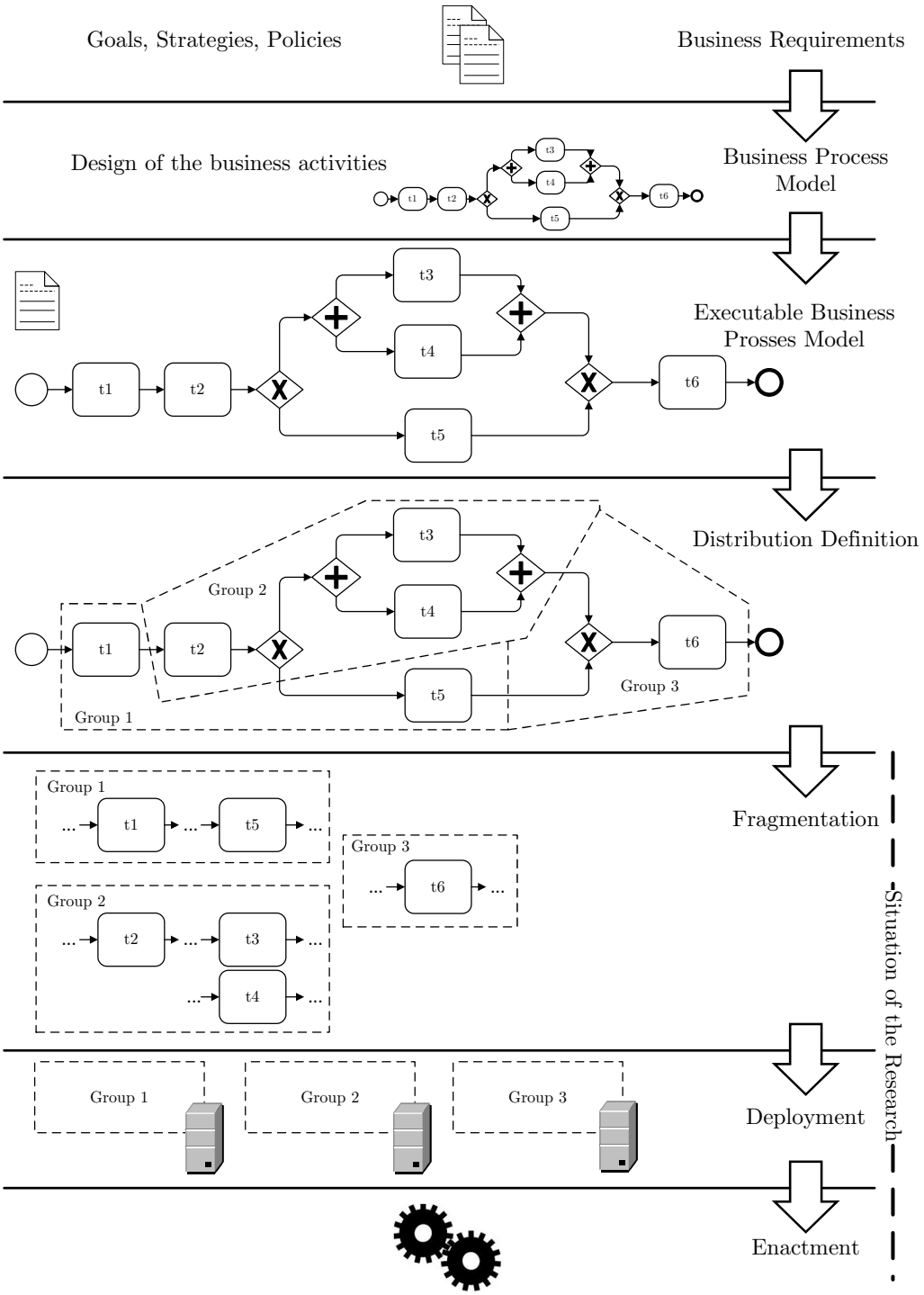


Figure 1: Research Situation

The approach is evaluated in three ways. First, we give a formal proof of similarity between the original designed process model and the distributed process execution, ensuring that the logical process execution after distribution is the same as originally modeled by the process modeler (Section 7.1). Second, we discuss the pros and cons of the approach by comparing the flexibility (Section 7.2, REQ4) with other distribution approaches and comparing the performance (and potential coordination overhead) with a (replicated) centralized execution environment (Section 7.3 and 7.4, REQ3). Section 7.5 further elaborates on any potential disadvantages.

Before presenting the details of the transformation and execution semantics, Section 2 first presents a running example and Section 3 provides the overall conceptual model of the autonomous distributed system for process execution.

2. Running Example

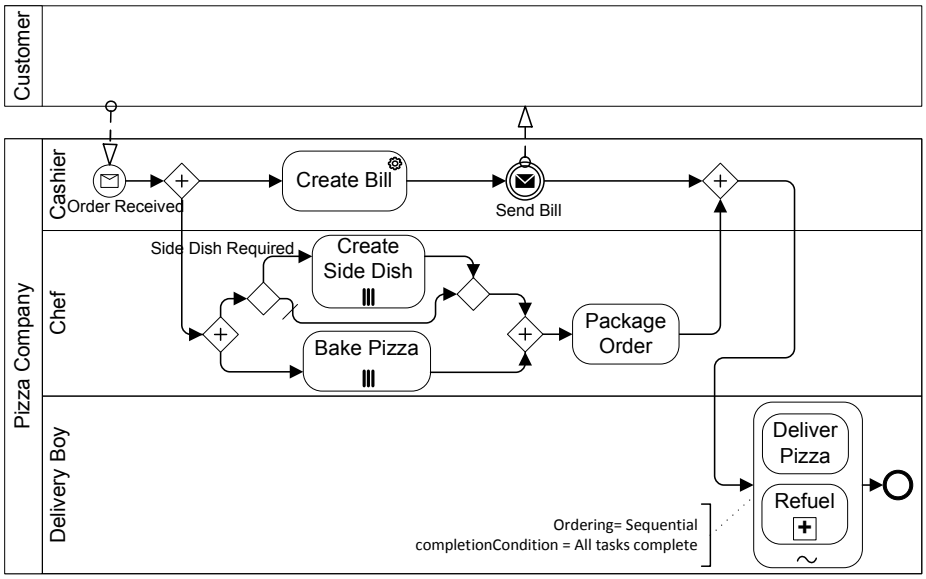


Figure 2: Pizza Delivery Example in BPMN (Object Management Group, 2010)

Figure 2 shows a small (fictional) BPMN (Object Management Group, 2010) process model for a pizza delivery company. It involves three parties: a chef who bakes the pizzas and creates, if required, side dishes, a cashier who receives the orders and arranges the payments, and a delivery boy who eventually delivers the pizzas and refuels (if necessary) the delivery truck in an arbitrary order. In the refuel subprocess (not shown) the fuel tank is checked, and if it contains less than 10% fuel, refueling is done.

The most frequently used process constructs are used in this process example (zur Muehlen and Recker, 2008; Recker, 2010) (simple sequence flows, exclusive and parallel gateways (split and join), lanes, a start and end event, an intermediate event which can communicate with other processes, human tasks and a service task), in addition to the more (executional-) complex ad-hoc subprocess. Although the example is kept

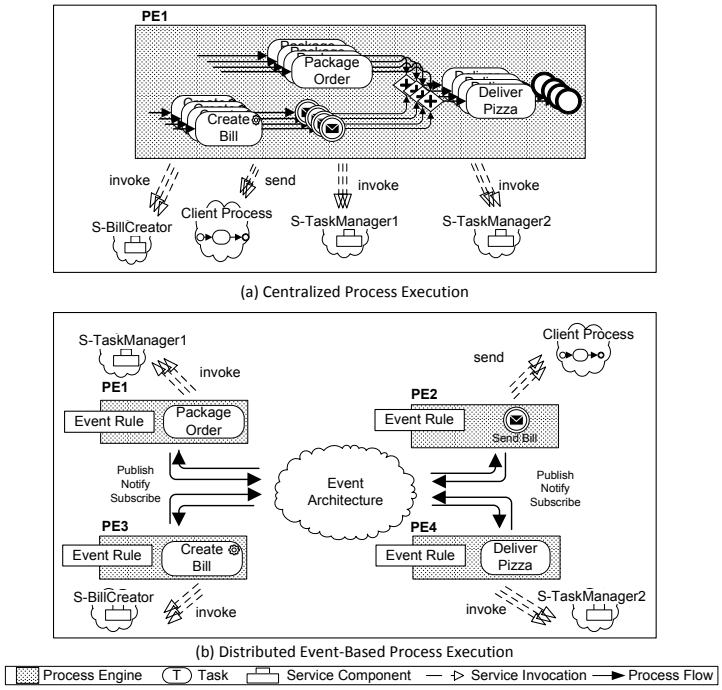


Figure 3: Centralized and Distributed Event-Based Process Execution

small for explanatory purposes, the proposed approach is not limited to these simple examples. More advanced process constructs, besides the ad-hoc subprocess, are also included in the transformation (see Section 4). Also note that this example omits data-flow considerations. Since this research is focused on the control flow of process execution, we assume data is transmitted along with the sequence flow. A short explanation of data handling in the distributed environment is given in Section 5. For a more elaborate discussion of data-dependencies in a distributed process architecture we refer to (Khalaf et al., 2008; Monsieur et al., 2012).

In the pizza company, a process engine is used to execute and control the process flow (see Figure 3a), task managers are employed to handle the inbox for manual tasks and one (web) service is used to create the bill, which can be invoked with technologies like SOAP and UDDI.

3. Distributed Event-Based Process Execution

In this section we present the overall conceptual model of distributed event-based process execution, which allows us to further refine the problem of developing an adequate transformation (Section 4).

Figure 3a shows in part the centralized execution of the pizza company's process. One process engine (PE1) is used to manage the entire process flow, whereby multiple process instances are controlled at the same time. As mentioned in the introduction, this central approach does not allow for the distribution of control and/or visibility of parts

of the process (one engine controls everything) and has scalability issues when dealing with large amounts of process instances. To overcome these managerial and technical issues, this paper looks at a process execution approach that uses not one, but multiple process engines to manage, in collaboration, the entire process flow. At deployment time a transformation algorithm takes as input the process flow as modeled by the process modeler and outputs different process model fragments. These fragments are deployed on different, dedicated process engines. Figure 3b shows the resulting distributed execution. Instead of one (or multiple duplicated) engine(s), multiple engines are used which differ both physically (location in the IT architecture) and logically (execute a different process model fragment) from each other (PE1-PE4). The global process execution for a certain process instance is performed by the collaboration of all the process engines running the different process model fragments.

To achieve this collaboration an event-based communication paradigm is used between the different process model fragment engines. In an event-based communication architecture, components communicate by generating and receiving *event notifications*, where an event is any occurrence of a happening of interest (Mühl et al., 2006). Components (the *publishers*) publish notifications into the architecture, where they are routed to other interested parties (the *subscribers*). This routing is done by an event service that keeps track of which entity is interested in which event and which entity is able to publish which event (content-based many-to-many routing). Collaboration between the different entities in an event-based system is accomplished by performing *publish*, *subscribe* and *notify* operations. A publish operation is a (event notification-) message from a publisher to the event service, a subscribe operation is a (subscription-) message from the subscriber to the event service and a notify operation is a (event notification-) message from the event service to the subscriber. In our case, the fragmented process engines are both publishers and subscribers of event notifications, with an event being a *past* happening in the PAIS *with a business meaning*, e.g. the completion of a business task, the arrival of an order request, the cancellation of a task, etc. Each notification hereby relates to a specific element in the original process model.

Decoupling of interaction partners is the main advantage of using event communication (Eugster et al., 2003). Due to its content-based routing, an event based architecture creates a highly flexible and scalable process execution infrastructure, where each node in the architecture becomes autonomous. The starting logic of a node is contained in the node itself (Mühl et al., 2006) (see Section 7.2). Eugster et al. (2003) defines the loose coupling advantages in a publish/subscribe system as space decoupling (the interacting partners do not need to know each other), time decoupling (interacting partners do not need to be active at the same time) and synchronization decoupling (interacting partners are not blocked during publication or notification of event occurrences). Note that the supporting entities in an event based architecture (the *cloud* in Fig. 3b) are also distributed and do not add another single point of failure. Many solutions exist that distribute the event based architecture itself (Carzaniga et al., 2001).

A consequence of using event-based communication is that each process model fragment should be complemented with the event types for which it needs a subscription (and thus a notification). This is called the *event rule* for the process model fragment. The event rule is a logical combination of event types that describes in which state the (business) environment should be before the process model fragment can start executing. The process model fragment is enabled if its event rule validates. For example, the

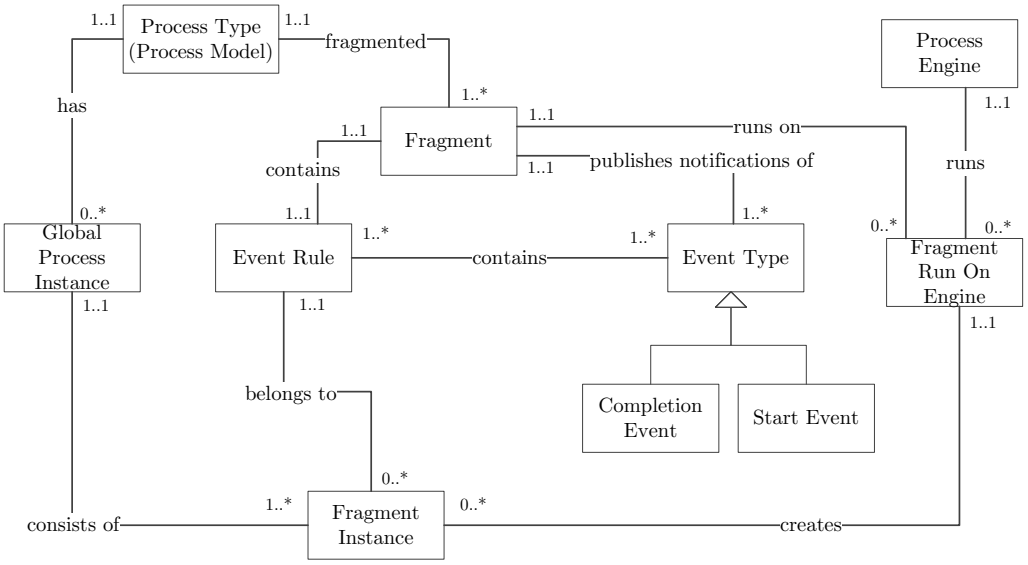


Figure 4: Components in Distributed Event-Based Process Execution

event rule for the ad-hoc subprocess from Figure 2 states that the subprocess can start when *Package Order* `and` *Send Bill* is completed (for the same process instance). In the resulting architecture (Figure 3b) publish, subscribe and notify operations are used to represent the sequence flow between the different process model fragments.

Figure 4 shows the major components of the distributed event-based process execution architecture. A process model is split into different process model fragments (at least one). Each process model fragment is executed on a (possibly unique) process engine. To enable load balancing at the fragmented node itself, multiple, duplicated engines per process model fragment are allowed. Each fragment contains an event rule which describes when the fragment can start executing. The event rule itself contains all event types for which subscriptions are needed for the corresponding fragment's engine in the publish/subscribe architecture. The fragment also publishes one or more events itself in the execution environment. A fragment can publish a *completion event* indicating the completed execution of a fragment or a *start event* indicating the start of the execution of the fragment (see Section 4). To handle the publications and subscriptions a process engine is contained in a publish/subscribe wrapper (not shown) which can communicate with the underlying event architecture and triggers the engine whenever necessary. Eventually the process fragment's engine will create process model fragment instances, publish event notifications and react to notifications, hereby creating a collaboration with others which results in the execution of the modeled, global process flow.

To reap the loose coupling benefits of using an event based architecture, the global process flow needs to be transformed into a distributed *event-based* process. The most important part in this transformation is finding the event rule for each process model fragment. This is described in the following section.

4. Process Transformation

After process modeling and full process specification (i.e. providing data links, partner links, etc.) a process model is ready to be deployed into the execution architecture. To be able to execute a process in the distributed event-based setting, the process model needs to be transformed into different process model fragments, with each a corresponding event rule. Before the event rule for each fragment can be calculated, the fragments in the process model have to be defined. In other words, a *distribution strategy* has to be chosen. A workflow can be split according to user-defined regions, according to workflow variants (Hallerbach et al., 2010), to decrease network traffic (Chafle et al., 2004), according to the domain a task belongs to (Kong et al., 2009), etc. The preferred distribution strategy depends on the requirements for process execution (technical or managerial). In order to enable any distribution strategy in the event-based fragment enactment environment, we do not choose a predefined fragmentation, but support any definition of process model fragments. This way, any distribution strategy can be used to realize a specific process model distribution goal.

In the next section, the fragmentation and distribution approach is explained. In the following sections, the transformation of the global process model to the event-based process model fragments is presented in two-fold. The first part is a basic transformation which transforms standard process elements: tasks (subprocesses, events, ...) and parallel and exclusive gateways (see Section 4.2). These process constructs support most process models (zur Muehlen and Recker, 2008; Recker, 2010) and languages (BPEL (minus the pick activity), UML Activity charts, BPMN Descriptive Conformance Subclass, ...). Secondly, the support for extra process constructs is described in Section 4.3. Other process entities than the ones described above require a higher degree of coordination, therefore needing more advanced event rules. In total, the transformation supports the *soundness preserving workflow patterns* 1-6, 8-10 and 12-20 as (re-)defined by YAWL[1] (van der Aalst et al., 2003), and therefore any modeling language incorporating these soundness preserving workflow patterns.

4.1. Fragmentation versus Distribution

In most decentralization techniques, fragments are defined in the original process model and these fragments serve as the units which are distributed in the IT architecture (Fdhila et al., 2009; Chafle et al., 2004; Bauer and Dadam, 1997). We however make a distinction between fragmentation and distribution. Figure 5 shows an example of this concept. The process model is first fragmented into task-level fragments[2]. Each task in the original process model becomes a small process on its own, complemented with an event rule stating when the fragment can start and a completion event indicating the completion of the fragment (task). After fragmentation, the fragments are grouped in a *distribution grouping* (step 2 in Figure 5). The distribution grouping defines how fragments are (physically) distributed in the architecture. The preferred distribution of activities (e.g. to minimize network traffic) is therefore represented by the distribution

[1] http://www.yawlfoundation.org/pages/resources/patterns.html

[2] Note that the *task-level* is defined by the original process modeler. A task can still contain multiple other instructions (e.g. a subprocess).

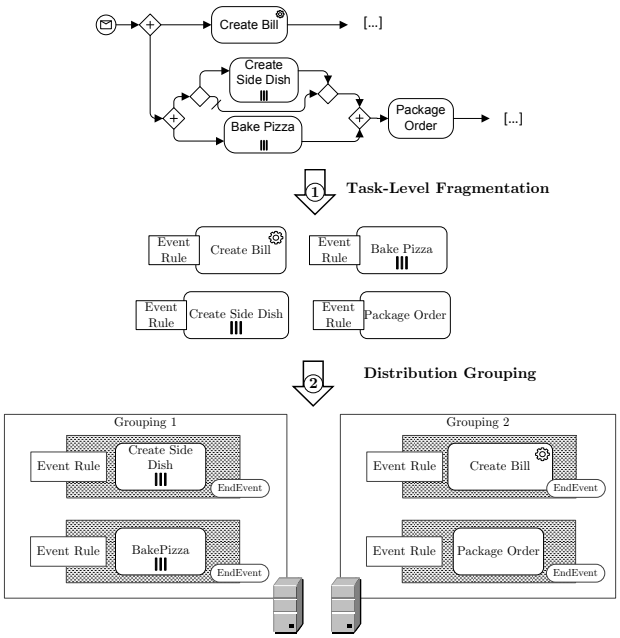


Figure 5: Fragmentation versus Distribution

grouping of different process model fragments. Task-level fragmentation therefore does not mean task-level distribution.

Creating fine-grained fragments and making a distinction between fragmentation and distribution has the following advantages:

Arbitrary distribution. There is no restriction on the creation of the distribution groupings. The fragments are used as building blocks for the distributed groupings and since a fragment consists of only one activity, an arbitrary distribution scheme can be created. Activities of the original process model can be grouped according to the same web service invocation, grouped according to the same human performer, grouped according to an optimization heuristic that e.g. optimizes network traffic, grouped according to user-defined regions, etc.

Flexible deployment structure. After deployment, the deployment structure (distribution grouping) can be changed ad-hoc. Fragments can be moved from one distributed entity to another with limited impact on the enactment environment. Due to the space decoupling, the publish/subscribe architecture can handle the physical movement of nodes (task fragments) in the architecture (Mühl et al., 2006).

Limited change impact when redeploying a changed process model. Since a task is used as unit of decomposition, each (business) task present in the original process model will give rise to a (completion-) event notification in the enactment environment. The fact that event notifications for each existing task are already present in the enactment environment limits the impact of a process model change on the runtime architecture. No new *synchronization messages* have to be created

when the structure of the process model is changed. See section 7.2 for a further discussion of the change impact.

A similar fragmentation and distribution technique is also applied by computer program compilers which perform automatic parallelization (Faraboschi et al., 2001; Tian et al., 2003). A program is split into a collection of single instructions after which these instructions are grouped to achieve an optimized instruction level parallelization.

In what follows, we focus on the creation and execution of *task-level fragments* and leave the distribution grouping out of scope (see also section 1.2).

4.2. Basic Transformation

Algorithm 1 Basic Transformation

Definitions

Process = $<T, G, SF>$
with T the set of tasks, G the set of gateways and SF the set of sequence flows
$SF \subseteq [(T \cup G) \times (T \cup G)]$
Event = $<id, Signal>$, with E the set of all events
Fragmented-Process = $<EventRule, t \in T, endSignal \in E>$
EventRule = $(\{event\} \vee \{event\} \vee ... \vee \{event\})$
= combination of events in Disjunctive Normal Form

Procedure split(Process P)
1: **for all** Task t ∈ P **do**
2: ER = eventRule(t)
3: FragProcess(t) = <ER,t,SignalOf(t)>
4: **end for**

Procedure eventRule(Task t)
5: $F = \{(x,t)|(x,t) \in SF\}$
6: **return** $\bigvee_{f \in F} eventRule(f)$

Procedure eventRule(SF (a,b))
7: **if** a = Task **then**
8: id++
9: **return** $<id, SignalOf(a)>$
10: **else if** a = StartOfProcess **then**
11: **return** $<id, a>$
12: **else if** a = XOR-Gateway **then**
13: $F = \{(x,a)|(x,a) \in SF\}$
14: **return** $\big(\bigvee_{f \in F} eventRule(f)\big) \wedge$
15: $conditions((a,b))\big)$
16: **else if** a = AND-Gateway **then**
17: $F = \{(x,a)|(x,a) \in SF\}$
18: **return** $\bigwedge_{f \in F} eventRule(f)$
19: **end if**

Procedure conditions(SF (a,b))
20: **if** $ExpressionOn(a,b) = default$ **then**
21: $F = \{(a,x)|(a,x) \in SF\} \backslash \{(a,b)\}$
22: **return** $\neg(\bigvee_{f \in F} ExpressionOn(a,b))$
23: **else**
24: **return** $ExpressionOn(a,b)$
25: **end if**

Algorithm 1 shows the procedures to split a global process into multiple fragments (in $O(n^2)$ time). Each resulting process model fragment consists of a starting rule, a task to execute and an (end) event to publish the completion of the task (see line 3). A starting rule for a split process consists of an event part (the event rule) and a user-defined conditions part (originating from XOR-splits in the global process). Finding the event rule for a split process equals finding, for a specific task, which preceding tasks in the process flow need to be completed before it can start its own execution. The algorithm finds these completion events by means of a depth-first search in the upward flow in the global process model. The event rule is transcribed as a logical expression, where an event in the expression is a *signal* which is matched with a publication event

of another process model fragment (see line 9). The conjunctions and disjunctions in the event rule match with the gateways preceding the process model fragment (see lines 12-18). If, for example, the fragment is preceded by a parallel gateway, the event rule will consist of a conjunction containing the publication events of the process model fragments preceding the parallel gateway. Only *tasks* will therefore be distributed, and the gateway logic preceding these tasks is included in the process fragment itself. This ensures that only events are published which have a business meaning (relate to a specific task in the original process model). Events indicate the completion of a task, the receipt of a message, etc. and not e.g. that a parallel split in a process path happened, that a conditional merge happened, etc.

The second part of the starting rule consists of user defined conditions originating from XOR-splits. These conditions are in conjunction with the event rule. Only when an event rule validates AND the respective conditions validate, then is the task in the process fragment enabled for execution. When searching for the completion events for the event rule, any condition encountered on an XOR-gateway is also stored in the starting rule (see lines 14-15). The language in which these conditions are expressed is situationally dependent. For example, some process engines rely on the Unified Expression Language (UEL)[3] to formally model conditions. The condition preceding the *Create Side Dish* task can be written in UEL as follows:

```
${SideDishRequired==true}
```

where `SideDishRequired` is a simple boolean variable which, depending on the process instance, can be true or false. In the rest of the paper we will write the expression in plain text (like is done on the process model in figure 2).

As an example, the basic event rule for the *Package Order* process model fragment is: ((BakePizzaCompleted $\wedge$ CreateSideDishCompleted) $\vee$ (OrderReceived $\wedge$ BakePizzaCompleted $\wedge$ No SideDishRequired)). A notification arrival of e.g. *BakePizzaCompleted* and *CreateSideDishCompleted* enables the event rule and enables the execution of the *Package Order* task. We refer to Figure 7 for more examples.

How the starting rule is concretely transcribed in the resulting process model fragments is dependent on the event architecture and the publish/subscribe wrapper used to contain each process fragment. Algorithm 1 is kept general, and only gives guidelines on how to build the split processes. Section 4.4 gives a concrete example of the transformation and the transcription of the starting rule, using BPMN2.0 as the process modeling language.

4.3. Extended Transformation

Although the basic fragmentation transformation supports the most frequently used process constructs, it is sometimes necessary to use a more advanced process modeling construct (like the BPMN ad-hoc process in Figure 2). The algorithm is therefore extended to allow most of the soundness preserving control-flow workflow patterns defined in (van der Aalst et al., 2003). Advanced process modeling constructs of BPMN2.0 however lack a sound and formal execution semantic. This makes it a cumbersome task to define a transformation for every BPMN construct. This problem can be solved by

[3] http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html

relying on the fact that these high level constructs can be translated to their operational semantics using a petri-net like formalism (van der Aalst, 1998). This creates a more precise execution semantic for each high-level concept, making it possible to also define a fragmentation for these high-level constructs. We choose to rely on YAWL (van der Aalst and Ter Hofstede, 2005) as the operational formalism. YAWL is a workflow modeling and execution language based on petri-nets, and contains a formal execution semantic. Many transformations are already defined which translate e.g. BPMN or EPC constructs to one or more YAWL entities, making them concrete and executable (Decker et al., 2008; Mendling et al., 2006; Ye and Song, 2010). YAWL is therefore a perfect candidate for the intermediate transformation (from high level concept to operational semantics). By using YAWL as basis for the extended transformation, most of the soundness preserving workflow patterns as defined by YAWL[4] are supported by the transformation: patterns 1-6, 8-10, 12, 16-20 in (van der Aalst et al., 2003)[5].

In the extended distribution algorithm, two constructs are added to support the distribution of more advanced workflow patterns. These are cancellation regions and condition-places (van der Aalst and Ter Hofstede, 2005). A cancellation region defines a process region that needs to be canceled (tokens are removed) when a certain task is started. A condition-place is a place in the process that indicates a certain state of the process, after which some actions can happen (see for example Figure 6). The semantics resemble these of a petri-net place. The cancellation concept is required to support the distribution of the cancellation workflow patterns (e.g. interrupting events in BPMN) and the condition-place is required to support the state-based workflow patterns (e.g. event-based gateway in BPMN). Note that the OR-join semantic (of YAWL and other languages) is omitted from the distribution algorithm, and therefore also the workflow patterns relying on the OR-join semantic. The OR-Join semantic is non-local and requires a high level of state-based knowledge of the current process execution, which contradicts with our loose coupled distributed environment (see the discussion in Section 7.5). Also note that transformations to YAWL as done by Decker et al. (2008); Mendling et al. (2006); Ye and Song (2010) add a *tau-* or *silent-activity* for some process constructs to the resulting YAWL process model. Since these silent transitions serve as *placeholders* (e.g. for an AND-Join-AND-Split construct), they are ignored in the transformation.

The added constructs require a higher degree of coordination between different process model fragments. Process model fragments need to notify each other, not only of their completion, but also of their consumption of a control flow token. In one case, this will disable the possible execution of other process model fragments (condition-place) and in other cases this will cancel the execution of already started fragments (cancellation regions). To cope with this extra coordination, besides the *event rule-events* and the *end-signal* event, two additional event-sets are added to the process model fragments. The *start-signal* element indicates the event notification that is published when the process model fragment starts its execution and the *cancellation* set contains all events for which the fragment needs a subscription and which, if notifications arrive, cancel the process fragment's execution. Both sets are optional and can be left empty (e.g. when a fragment

[4]http://www.yawlfoundation.org/pages/resources/patterns.html

[5]Note that patterns 13 and 14 (multiple instances with synchronization) are supported if the pattern is transformed to a combination of XOR and AND gateways, with an event queue enabling a counter for the multiple instances as described in (van der Aalst et al., 2003).

Algorithm 2 Extended Transformation

Redefine
Process = $<T, G, SF, CP, cancel>$
with CP the set of Condition-places
$SF \subseteq \big[[(T \cup G) \times (T \cup G)] \cup [(T \cup G) \times CP] \cup [CP \times (T \cup G)]\big]$
$cancel : T \rightarrow \{T\}$ specifies the tasks which cancel the execution of a given task
Fragmented-Process = $<EventRule, t \in T, endSignal \in E, startSignal \in E,$
$cancellation \subseteq E>$
rewrite Event = ◊ Event
rewrite BasicEventRule = ◊ [BasicEventRule ∧ □ ¬ StartSignalOf(t)]

Addendum eventRule(Task t)
1: **if** a = ConditionPlace **then**
2: $F = \{(x, a) | (x, a) \in SF\}$
3: $N = \{x | (a, x) \in SF\}$
4: **return** $\Diamond\Big[\big(\bigvee_{f \in F} eventRule(f)\big) \wedge \Box\neg\big(\bigvee_{n \in N} startOf(n)\big)\Big]$
5: **end if**

Procedure startOf(ProcessElement t)
6: **if** t = Task **then**
7: add StartSignalOf(t) to *startSignal* set of t
8: **return** StartSignalOf(t)
9: **else**
10: $N = \{x | (t, x) \in SF\}$
11: **return** $\bigvee_{n \in N} startOf(n)$
12: **end if**

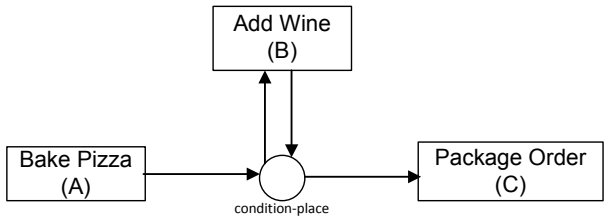
Addendum split(Process P)
13: **for all** Task t ∈ P **do**
14: [...]
15: **if** cancel(t) != null **then**
16: **for all** x ∈ *cancel*(*t*) **do**
17: add StartSignalOf(x) to *cancellation* set of t
18: add StartSignalOf(x) to *startSignal* set of *x*
19: **end for**
20: **end if**
21: **end for**

is not canceled by others or no start event needs to be published).

As a result of the inclusion of the condition-place, event behavior becomes dynamic: over time, an enabled task can be disabled again. To incorporate this extra behavior, events are appended with a time concept, which allows the expression of the event rule in Temporal Logic (LTL) (Pnueli, 1981). Note that we use LTL as a formal notation. Any rule language supporting temporal constraints can be used in an actual implementation of the algorithm (e.g. RAPIDE in CEP (Luckham, 2002)).

The event rule will be evaluated on the stream of events received by the process model fragment (which can grow in time). Figure 6 gives an example of why the temporal aspect is needed. The figure shows the milestone pattern (van der Aalst et al., 2003), with three tasks A, B and C and one condition-place. The example depicts a process where only at a state right before C, a task B is able to execute (e.g. for the pizza company, allowing the addition of a wine bottle to the order providing the *package order* task has not yet started). Task B and C are only enabled for execution when the place contains a token. A sample trace of event notifications accepted by task C is presented on Figure 6. The status of task C can change over time, where task C is only enabled for execution when *B-Completed* or *A-Completed* is the last event received. A temporal aspect is needed to capture this information, with which the following (LTL) start rule can be constructed for task C:

$$\Diamond\big[(\text{A-Completed} \vee \text{B-Completed}) \wedge \Box\neg(\text{B-Started} \vee \text{C-Started})\big] \quad (1)$$



Time	Trace of notifications accepted by C	Status of C
t	: A-Completed	(Enabled)
t+1	: A-Completed; B-Started	(Disabled)
t+2	: A-Completed; B-Started; B-Completed	(Enabled)
…		
t+n	: A-Completed; …; B-Completed; C-Started	(Started)

Figure 6: Example of an event execution trace

with ◊ the *eventual* temporal logic operator and □ the *always* temporal logic operator (Pnueli, 1981). The rule states that eventually *A-Completed* or *B-Completed* should have happened (the condition-place contains a token), but neither *B-Started* nor *C-Started* happened (the token is not yet consumed). If the event rule evaluates to true, at any time, on the event trace accepted by fragment C, then the fragment is enabled and ready to be executed.

Algorithm 2 shows the extra transformations needed to incorporate the condition-place and the cancellation regions. First, the basic event rule from Algorithm 1 is rewritten to incorporate LTL semantics. The condition-places are then transformed into an LTL rule similar to equation 1 (see line 4). Cancellation regions are supported by including each event which cancels the process fragment in the *cancellation* set of that process fragment (see lines 15-17). At the same time, wherever the need, start events are included in the process fragment (see lines 7 and 18).

4.4. Prototype Implementation and Example

For a proof of concept, we implemented the algorithm in java/EMF[6] using BPMN2.0 as the process language for both input and output models. An advantage of describing the process model fragments in the same language as the global model, is that existing process engines (and modeling/visualization tools) supporting the global model can also execute the distributed process flow, as long as they are contained in a publish/subscribe wrapper which handles the event communication. As output, xml files conforming to the BPMN2.0-XML structure are generated, each containing a process model fragment description (copied from the original model). The xml-files are extended to include the event-sets: *endSignal*, *startSignal*, *cancellation* and the *event rule*. The basic event rule is captured in a simple set containing disjunctions, which again contains conjunctions of events and user defined conditions (Disjunctive Normal Form). For the LTL event rules, the LTL notation of the LTL-Checker included in the Prom framework (van Dongen et al., 2005) is used. An example of the output bpmn-xml file for the process model fragment *Package order* is shown in Figure 8.

[6]Available at http://code.google.com/p/debo/

As an example, the pizza delivery's process model fragments are shown in Figure 7, each with their corresponding event rules and event sets. Note that, as said above, high level (BPMN) process constructs are not directly supported. The ad-hoc BPMN subprocess is first transformed to its operational semantics (in YAWL, using the technique of Ye and Song (2010)), which is then used to calculate the event rule of each individual part. For example, the event rule for Deliver Pizza states that it can only start when the previous two tasks, Send Bill and Package Order, are completed (part(1) in Figure 7) *and* that the condition-place should contain a token (i.e. the Refuel task is not already started, see part (2)). The conjunction between part (1) and (2) is the result of the AND-Gateway rule in Algorithm 1 (lines 16-18), where part (2) itself is the result of the condition-place rule in Algorithm 2 (lines 1-4). Also note that *Side Dish Required* and *No Side Dish Required* are no events but user defined conditions (see Section 4.2). The conditions are copied from the original process description (expressed in a formal language) and checked on the data received by the process (see Section 5.3 on data handling).

The generated xml files (as in Figure 8) can be used to deploy and execute the process model in a distributed event-based manner. This is described in the following section.

5. Process Execution

After the transformation, each process model fragment is deployed to a dedicated fragment engine, where each engine is contained in a wrapper that handles the publish/ subscribe communication. These fragment engines can hereafter be distributed in the enactment environment. Section 5.1 describes the execution architecture, section 5.2 describes how a fragment is triggered for execution and section 5.3 shortly describes how data can be handled.

5.1. Execution Architecture

Figure 9 positions the distributed process execution and its respective publish/subscribe middleware in a service oriented architecture (SOA), realized by an enterprise service bus (ESB). The ESB is an enabler for SOA by providing a connectivity layer between different services (Chapell, 2004). The publish/subscribe middleware is a part of the infrastructure services which provides the routing and subscription facilities for the event messages (originating from the process model fragments). Each process fragment engine is a (small) composition service, which coordinates, in our case, a single business task (e.g. invoking one or more business services in the ESB). Additionally, management and monitoring services can be added which allow for the observation and administration of the distributed process execution.

The internal workings of a fragment engine embedded in a publish/subscribe wrapper are shown in Figure 10. The wrapper accepts event notifications and creates fragment instances which are linked to their respective event rule (also see Figure 4). An already existing process engine (e.g. BPEL, YAWL or BPMN engine) is also included in the fragment engine to handle the actual execution of the process model part once the instance is triggered by its event rule. The responsibility of the fragment engine is to manage the state of the event rule of each fragment instance and trigger the execution of any instance for which the event rule validates. The embedded process engine takes care of

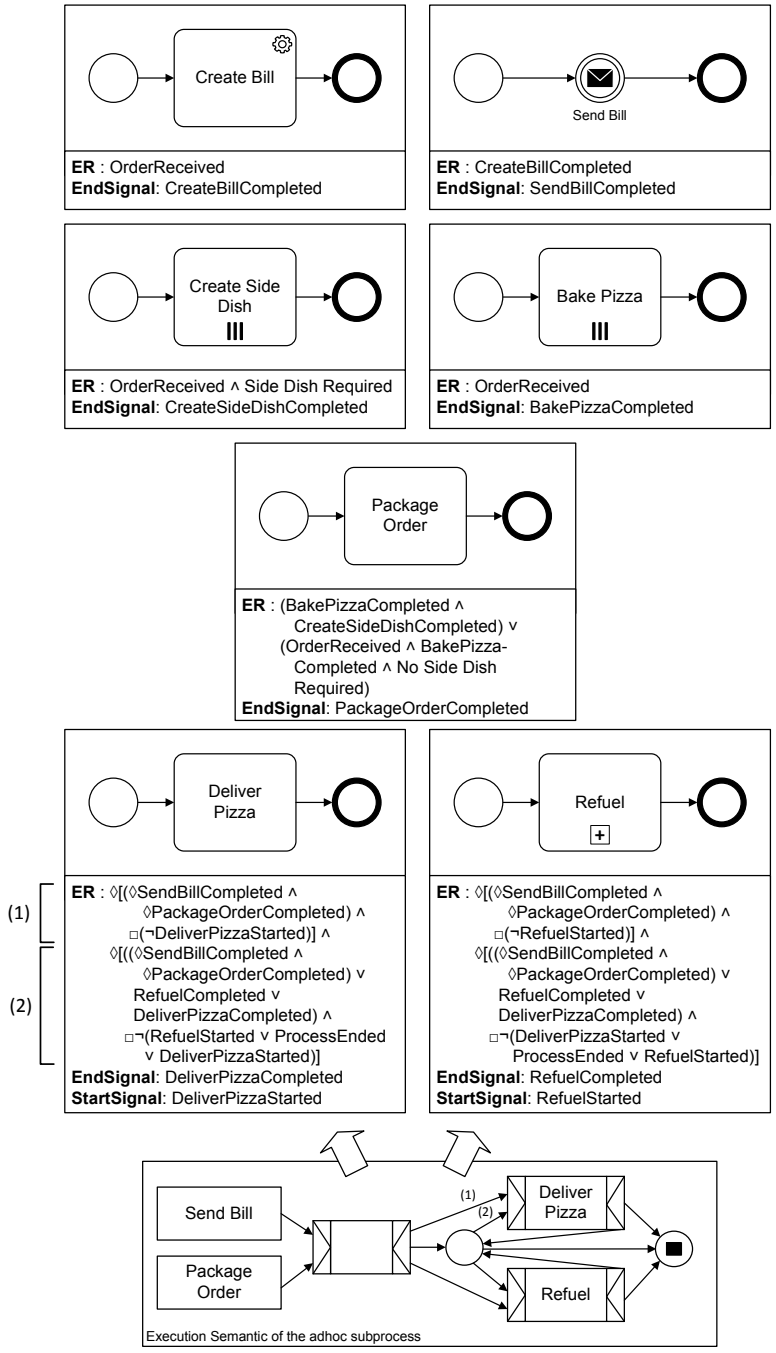


Figure 7: The process model fragments, their event rules (ER), End Signals and Start Signals of Figure 2

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:pubsub="http://www.kuleuven.be/pubsub" targetNamespace="Example">
 <process id="p1" isExecutable="true">
   <extensionElements>
     <pubsub:basicEventRule>
       <pubsub:conjunction>
         <pubsub:event id="bpc1">BakePizzaCompleted</pubsub:event>
         <pubsub:event id="csd1">CreateSideDishCompleted</pubsub:event>
       </pubsub:conjunction>
       <pubsub:conjunction>
         <pubsub:event id="or1">OrderReceived</pubsub:event>
         <pubsub:event id="bpc1">BakePizzaCompleted</pubsub:event>
         <pubsub:condition>!SideDishRequired</pubsub:condition>
       </pubsub:conjunction>
     </pubsub:basicEventRule>
     <pubsub:endSignal id="id1">PackageOrderCompleted</pubsub:endSignal>
     <pubsub:startSignal/>
     <pubsub:cancellation/>
   </extensionElements>
   <userTask id="t1" name="Package Order">
     <incoming>sf1</incoming>
     <outgoing>sf2</outgoing>
     […]
   </userTask>
   <startEvent id="sev1" name="">
     <outgoing>sf1</outgoing>
   </startEvent>
   <endEvent id="eev1" name="">
     <incoming>sf2</incoming>
   </endEvent>
   <sequenceFlow id="sf1" name="" sourceRef="sev1" targetRef="t1"/>
   <sequenceFlow id="sf2" name="" sourceRef="t1" targetRef="eev1"/>
  </process>
</definitions>
```

Figure 8: The bpmn-xml file for the Package Order fragment

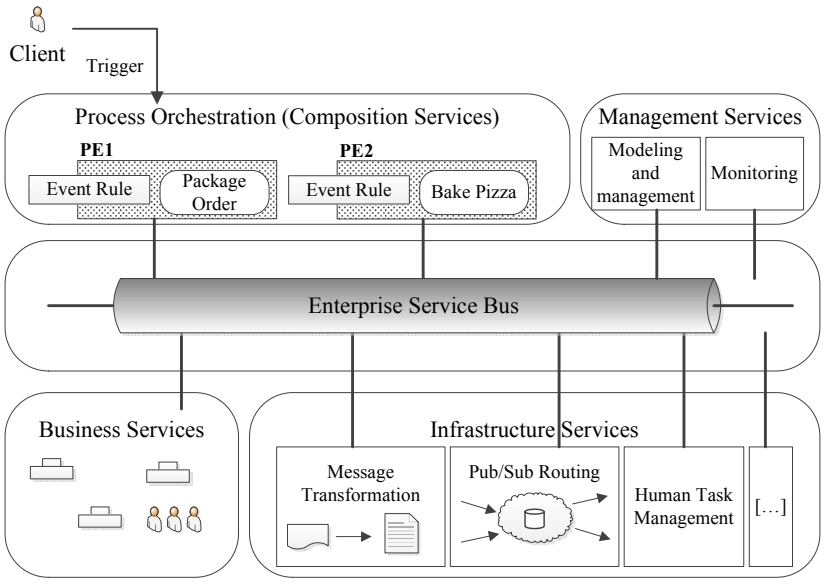


Figure 9: Distributed event-based process execution in the Enterprise Service Bus

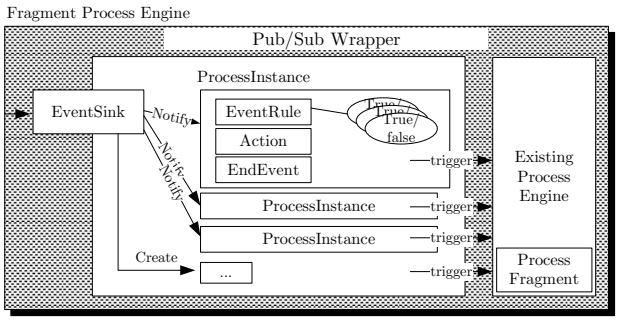


Figure 10: Fragment Process Engine

the actual execution of the process model part described in the fragment (e.g. invoke a web service).

The data payload (content) of an event message in the architecture should minimally consist of two things, one is the indication of the task it represents, and the other is a process instance id, indicating for which (global) process instance an action has been performed. The latter attribute is necessary not to loose the coupling between the process instance and the action performed (also see the link between fragment instance and global process instance in Figure 4).

5.2. Execution Semantics

In what follows, four steps are described which explain the execution of the dedicated fragment engine. Note that a distinction is made between the use of the basic event rule (Section 4.2) and the use of an LTL event rule (Section 4.3). This is done because checking a temporal logic rule on an event trace is significantly more complex than checking a basic first order logic rule (Sistla and Clarke, 1985). Where possible, the basic event rule is used.

Figure 11 shows the possible states of a fragment instance in the fragment engine. Deploying and executing a process model fragment means that the following steps need to be done:

1. (a) The publish/subscribe wrapper issues subscriptions to all events in the process fragment's description (i.e. all events found in the basic event rule or LTL event rule). Similarly, subscriptions are issued for each event in the cancellation set of the fragment's description.
   (b) If the event architecture requires advertisements to be made for each event published by an entity, the wrapper can create advertisements for events found in the *end-signal* and *start-signal* sets. For example, an advertisement for the `BakePizzaCompleted` event for the *Bake Pizza* fragment is issued.
2. After subscription, the wrapper is ready to accept event notifications. When a notification arrives at the publish/subscribe wrapper, it is routed to the corresponding process fragment instance, using the process instance id attribute found in the notification payload (see the *notify* links after the *EventSink* in Figure 10). If the instance does not yet exist, a new one is created (initially in state *evaluating*, see Figure 11). Next,

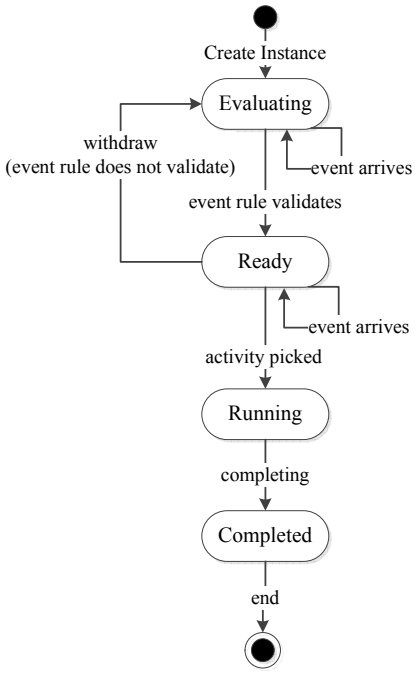


Figure 11: State machine of a process instance

(a) for a *basic event rule* the event notification is matched to the correct event in the event rule, which will become enabled;
(b) for a *LTL event rule* the received event is added to the event trace of the fragment instance.

3. After each notification receipt, the event rule is evaluated. If the rule evaluates to true (i.e. all events in a conjunction in the basic event rule are enabled, or the LTL-rule evaluates to true on the event trace), the fragment instance's execution is triggered. The fragment instance state will move from *evaluating* to *ready* (see Figure 11). In the *ready* state, the fragment instance can be picked for execution. When it is picked, the native BPMN/BPEL/... process engine takes over and executes the actual work (invoking a service task, invoking another business process, enabling a manual task, sending a message, ...). Note that for automated tasks, the state will instantaneously move from ready to running. Only for manual tasks/ fragments, the fragment needs to be picked for execution. It is possible that a user picks another task to execute, which again disables the current fragments execution (as in Figure 6), hereby moving the fragment instance state back to *evaluating*.
If the fragment is picked for execution and changes its state to *running* the publish/ subscribe wrapper sends an event notification indicating the start of this process model fragment[7] (if it is included in the fragment's description). At the end of

[7]To enable concurrent execution, the sending of start-event notifications should be handled with a distributed semaphore Schneider (1982), this ensures that e.g. task B and C in Figure 6 never start at the same time, due to network delays of sending the start-event message.

process execution, a notification is published by the publish/subscribe wrapper to signal the end of this fragment's instance, with the corresponding `event-id` and `process-instance-id` included as attributes.

4. Received cancellation events are handled according to the status of the fragment instance. If the instance is already started, the publish/subscribe wrapper triggers the cancellation of the process instance. If the instance is not yet started, any events for the corresponding process instance will be ignored. Ignoring a notification can be handled by the publish/subscribe wrapper itself, or a new subscription can be issued to the event architecture, unsubscribing for any event associated with the corresponding process instance. This ensures the non-execution of the canceled process model fragment.

The published completion event from a process instance, sent in step 3, is routed through the event architecture, and picked up by other interested process model fragment engines. These will handle the event notification again with the steps described above. Eventually, the combined execution of all the dedicated process engines have achieved the global execution of the entire, designed process flow.

Note that the above steps require the publish/subscribe wrapper to handle the event matching and rule evaluation. Some event architectures provide the ability to perform these actions on the architecture itself (content-based filtering), hereby relieving the process engine of the matching work, creating more lightweight engines. Similarly, Complex Event Processing (Luckham, 2002) engines can be used to check the event rules, therefore already capturing the coordination logic in the event bus.

The engine is implemented in Java[8] using the Siena wide area event notification service Carzaniga et al. (2001) as the event service architecture, the Activiti BPMN2.0 process engine Alfresco (2012) to interpret and execute the actual process model fragment and the LTL model checker included in the ProM framework van Dongen et al. (2005) to evaluate the LTL rules.

5.3. Handling data

To be complete, data also needs to be transmitted through the event architecture. For example, the cook in Figure 2 needs to know which and how many pizzas he needs to create. The process engine handling the *Bake Pizza* task has to have access to the pizza order received at the start of the global process. Furthermore, when user defined conditions are included in an event rule, e.g. the `${SideDishRequired==true}` condition, the value of the included variable should be available to the process fragment (to be able to evaluate the event rule).

There are three methods to incorporate data into the distributed execution (Li et al., 2010). A first option is that, similar to the event-id and the process instance id, data objects are included in the payload of the event message. Each event message is appended with the data output generated by the current task execution. This has as advantage that the process data is available for every fragmented process engine, but creates an overhead because of the sometimes unnecessary data included. A second option is that data is handled in a publish/subscribe fashion (Li et al., 2010), where each entity holds a local

[8] Available at http://code.google.com/p/debo/

copy of the data object. If a data object is modified, a notification is published describing the changed data. Each interested party will receive this notification and update its local copy. In contrast to the first data-handling technique, only necessary distributed entities will receive data-updates. Because of the local copies, however, discrepancies can arise between the copies when not implemented correctly or errors occur (e.g. a local node did not receive a data update notification and is therefore still working with outdated/ incorrect data). A third option is to have a central data-storage which can be used to manage the process data. This storage is accessible by every distributed process engine, which are able to request data objects, update them and write them back to the central storage. An advantage is that data is only requested when necessary and that data stays up-to-date in the central storage. However, this again introduces a central point in the distributed architecture and hereby a single point of failure.

Because each technique has its own advantages and disadvantages, the choice on which data handling technique to use is situationally dependent. Section 7.3 gives a brief evaluation of the approaches. For a further elaboration on distributed data management, we refer to Khalaf et al. (2008); Monsieur et al. (2012).

6. Process Evolution and Version Management

The previous sections described the distributed event-based execution architecture for the execution of a process model predefined at design time. Processes however evolve over time and the execution environment should adequately support these changes (van der Aalst and Jablonski, 2000; Weber et al., 2009). In the distributed event-based environment, we identify two types of runtime process model changes: *top-down process model changes* and *bottom-up process model changes*. The top-down changes are process model changes originating from the original (global) process model. The new process model needs to be deployed in the runtime enactment environment, meaning that the impact of the change on the specific fragments and their event rules has to be investigated. A bottom-up change is a model change performed locally at a specific fragment, which consists of adapting an event rule and propagating this change to other runtime fragments (without knowledge of the original global process model). Both change types are described in full detail in (Hens et al., 2013) and (Hens et al., 2014) respectively. In this section we provide the reader with a brief summary of change and version support in the distributed environment. For more detail we refer to Hens et al. (2013).

6.1. Versioning in the Distributed Enactment Environment

Processes change over time, which means that multiple process model versions (schemas) can exist for one process type (Reichert et al., 2003). The relationship between process type, process schema and process instance is shown in Figure 12. Each schema represents another version of the process type. The process management system should support the evolution of the process by allowing the redeployment of a changed process model into the already running execution environment (adding new process model versions). After redeployment, any new process instance creation requests are handled with the new process schema. Moreover, to be completely flexible the system should support the migration of already running process instances to the newly deployed process model version (Rinderle et al., 2004).

In the distributed environment, a similar reasoning is applied. Figure 13 shows the versioning situation in the distributed environment. A process type has different schemas, each representing a different version of the process type. As explained in the previous sections, a process schema is transformed into different fragments at deployment time. Fragments are however not duplicated into the distributed environment according to the schema they correspond to. Fragments may overlap across schema versions. Instead of duplicating the existing process fragment and assigning it another event rule, the same fragment (process engine) is used and a new event rule is *added* to this fragment. Hence, a single process fragment can contain multiple event rules, each representing another schema version. In distributed execution, *the possibly multiple event rules of a process fragment represent the different versions of a process type* in the actual runtime environment. An advantage of this approach in which fragments are reused and event rules are updated, is that only fragments for which the event rule changes need to be updated and redeployed. Fragments where nothing changes can be left untouched and running in the architecture (keep using the same event rule). In Figure 13 for example, fragment 2 still uses its original event rule and serves all three process schemas. The benefit of this approach is evaluated in Section 7.2.

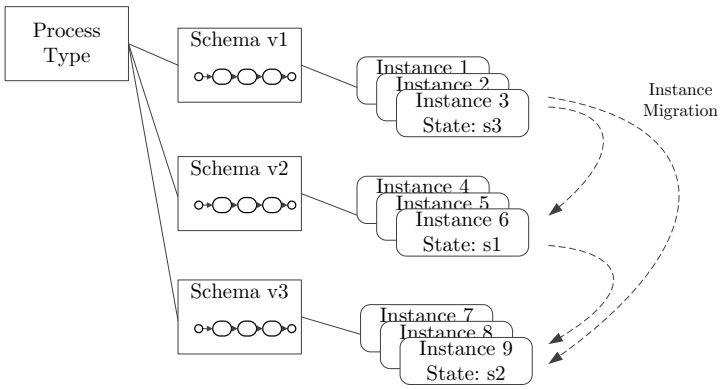


Figure 12: The relationship between a process type, its schemas and the schema instances in a centralized execution environment.

6.2. Process Evolution Protocol

Process evolution, meaning that a new process model version needs to be deployed in the runtime environment, is supported in the distributed architecture by the follow protocol (Hens et al., 2013):

1. Determine the change region, based on the old and new process model version (van der Aalst, 2001);
2. Suspend the enactment of the process fragments impacted by the change;
3. Determine the non-migratable process instances;
4. Fragment the new process model;
5. Update the event rules of each process fragment with a changed event rule;
6. Resume process execution.

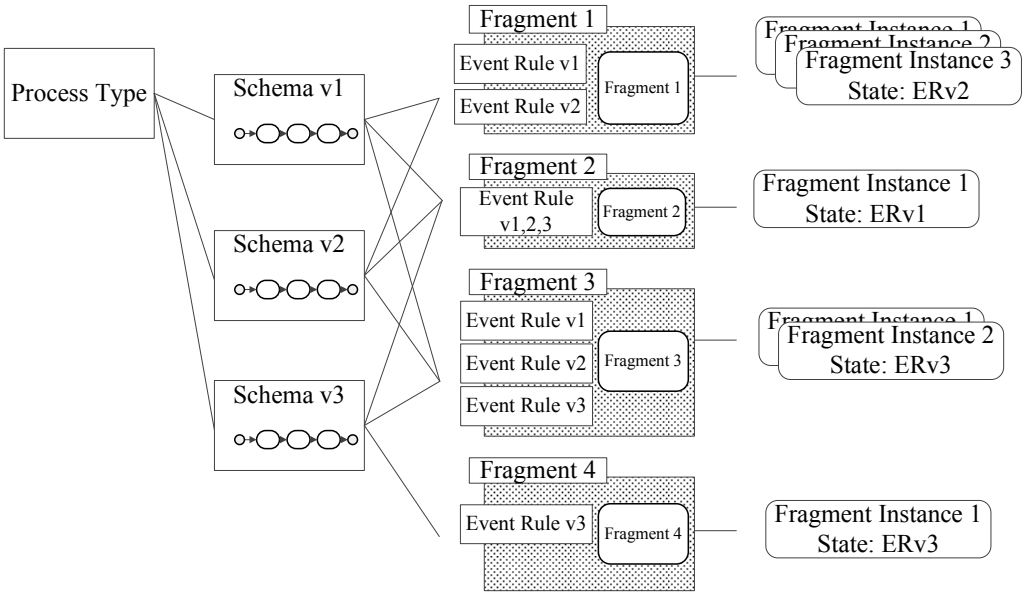


Figure 13: The relationship between a process type, its schemas the process fragments and the fragment instances in the distributed execution environment.

The first step is done to be able to determine which running process instances can migrate to the newest process model version (Rinderle et al., 2004). To this end, the change region technique, introduced by van der Aalst (2001) is used. The change region defines a region of *non-migratable states* in the old and new process model. Any process instance which resides in a state inside the change region is not able to migrate to the newest process model version. The advantage of using a graph based correctness criterion (Rinderle et al., 2004) for process instance migration is that we can precisely define the affected fragments in the runtime environment, increasing the control for suspension and redeployment (see step 2 and 3). In the second step, a minimal set of fragments is suspended in the enactment environment. This means that only fragments which are affected by the change (either have a changed event rule, or require a state inspection in step 3) are suspended. All other fragments can continue their execution. In the following step, all fragments which reside inside the change region are inspected and asked to return all process instances that are currently running on the fragment. These are the process instances which are not able to migrate to the newest process model version. In steps 4 and 5, the new process model version is fragmented and each fragment which has a new event rule is updated. When a fragment needs to update its event rule, it adds the new rule to a *list of rules* (see Figure 13). In this list of rules, one rule is made the *current rule* and all others are linked to a set of process instance IDs from process instances that require the use of that specific (older) rule. Upon receipt of an update message with a new event rule, the fragment links its current rule to the IDs of all non-migratable instances (collected in step 3) and makes the new event rule the current one. Versioning at the level of the fragments is therefore supported by allowing multiple event rules per fragment and linking each rule to the process instances requiring

the use of that specific event rule. This can be done, as all the non-migratable process instances (that still need the old rule) are collected in step 3. Because now a fragment has multiple event rules (versions), step 3 from the execution semantics of the fragment engine changes slightly (see Section 5.2). Upon arrival of an event notification message, it is checked if this event should be handled with the current rule or an old rule. An old event rule is used if the instance id included in the event notification message is also included in an ID-set of the rules-list. In the last step of process evolution, all suspended fragments are resumed and normal execution can continue.

The previous process described the top-down changes, these are changes that are made to the original global process model. Bottom-up changes use a similar approach of linking event rules to process instances. However, an additional check has to be implemented to make sure that a change made to a specific event rule does not break the global process execution. For example, changing a rule `A OR B` to `A AND B` can create a deadlock in the global process execution. This additional check is explained in more detail in (Hens et al., 2014).

7. Evaluation

To evaluate the transformation and event-based distributed execution of the process model, we look at three elements in the process transformation and execution. First, the correctness of the transformation is validated. The behavior of the distributed process fragment execution is compared to the behavior of the original process model, and it is checked if these are the same. Second, we look at the flexibility benefits of using an event-based distribution and compare these to a request based distribution (REQ 4 in Section 1). Last, to validate the technical benefits, the performance of the publish/subscribe process execution is tested and compared to centralized process execution (REQ 3 in Section 1). Note that REQ1 is satisfied because an automatic creation of the fragments is provided (Section 4) and REQ2 is satisfied by design (Section 4 and Section 3). The shortcomings and difficulties of the distributed event-based approach are discussed in the last section.

7.1. Formal Evaluation

We check if the distributed event-based process execution after transformation exhibits the same behavior as the originally modeled process in two ways. First, we check if the event rules generated by the transformation are correct on the execution traces accepted by the originally designed process model. This proves the correctness of the transformation (see Section 7.1.1). Second, the running event-based distributed architecture is compared with centralized execution. Both execution architectures should exhibit the same behavior and should be indifferent for an outside observer (behavior equivalence). This is demonstrated in Section 7.1.2.

7.1.1. Correctness of the transformation

The correctness of the transformation is validated by checking the LTL event rules on all possible execution paths of the supported workflow patterns (van der Aalst et al., 2003). For each of these patterns, execution traces are generated[9] and the LTL-event

[9]When indefinite loops exists in a workflow pattern, maximum 5 replications are taken for that loop.

rules are created using the algorithm described in Section 4. The execution traces depict the original (desired) behavior of the workflow pattern and the LTL-event rule depicts the behavior of the distributed execution (of a given process fragment). Given a pattern's execution traces, a distributed task or process element T and its LTL-event rule F, the following properties should evaluate to true:

1. (liveness) $\forall \pi : \pi_{0..i} \vDash F \rightarrow \exists \tau : \tau_{0..i} \vDash F \wedge \tau_{i+1} \vDash T$
2. (safety) $\forall \pi : \pi_i \vDash T \rightarrow \pi_{0..i-1} \vDash F$

with: π and τ execution traces of the workflow pattern, $\pi_i \vDash F$: the sub-trace with elements at time $i..\infty$ which satisfies the formula F and $\pi_{i,j} \vDash F$: the sub-trace with elements at time $i..j$ which satisfies F.

The liveness property states that if at some time the event-rule evaluates to true, there exists a trace (the same, or another) where the next element in the execution trace is the execution of the distributed process fragment. It is thus correct that a process fragment executes when its event rule is enabled. The second property states the safety property: when the process fragment executes, its event rule should be enabled at a time just before its execution.

The two properties have been checked and validated on the execution traces of the supported workflow patterns using the LTL checker included in the ProM framework (van Dongen et al., 2005)[10].

7.1.2. Correctness of the execution architecture

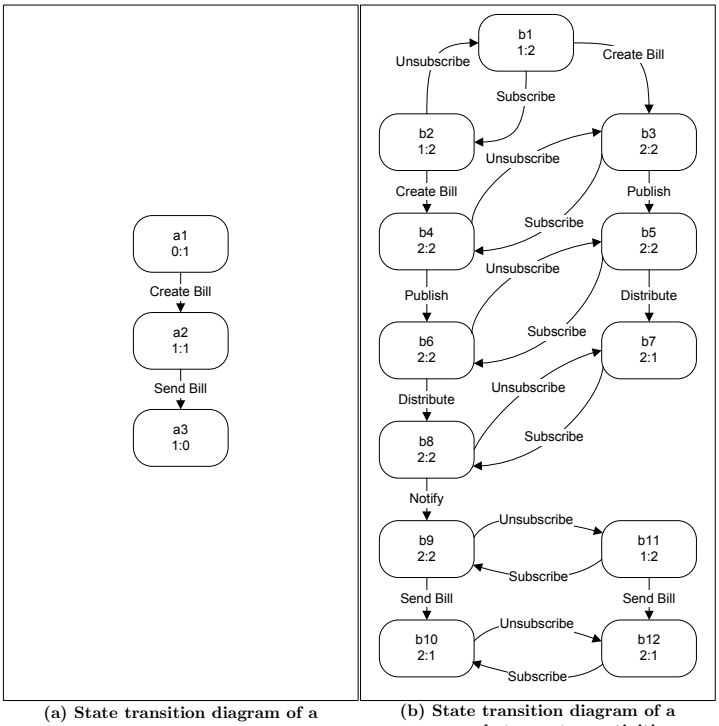
To be completely non-intrusive, the publish/subscribe execution architecture should perfectly mimic the behavior of the centralized process execution architecture. To check this equivalence, a petri-net model is developed representing the publish/subscribe architecture. This model allows us to reconstruct the distributed execution model, including all publish/subscribe operations. With this model, state charts can be generated, enabling behavioral equivalence checks with the original process model.

The petri net model is described in full in a technical report (Hens et al., 2011). As an example for the equivalence checks, the state charts for a sequence between activities *Create Bill* and *Send Bill* for the original model and the publish/subscribe execution of this sequence are shown in Figure 14. The equivalence of these two models can be checked using the notion of branching bisimilarity defined by Basten (1996), with the publish/subscribe operations as silent steps. The two workflows (with and without the publish/subscribe connection) are observationally equivalent and exhibit the same behavior. As outlined in detail in the technical report, the same conclusion can be drawn for the other transformation elements. Distributing and running a process model with a publish/subscribe communication scheme does not structurally change the business process execution itself and stays, for an observer, equivalent with the original centralized process execution.

7.2. Flexibility Benefits

Classical process distribution approaches use a directed request based coordination between the specific process fragments (Khalaf et al., 2008; Fdhila et al., 2009). Compared to this, using a publish/subscribe communication style creates a couple of flexibility

[10]Since the LTL checker does not support sub-traces, preprocessing is done to first generate the sub-traces on which the formulas are checked.



(a) State transition diagram of a sequence between two activities Create Bill and Send Bill

(b) State transition diagram of a sequence between two activities implemented with pub/sub operations

Figure 14: State transition graph of a sequence between transitions *Create Bill* and *Send Bill*, without and with the use of a publish-subscribe connection

benefits with respect to the request based approaches (Mühl et al., 2006; Eugster et al., 2003): a flexible process runtime with smaller change impact, flexible process deployment and a runtime plugin ability for e.g. process monitoring and management. Note that these flexibility benefits do not appear with respect to a centralized process execution. Indeed, process decentralization adds additional coordination overhead (see also section 7.3). In the centralized approach, all information is available in a single process model (e.g. for process evolution, see section 6). The following is therefore a discussion of the flexibility benefits of the event-based distributed approach with respect to other process distribution approaches:

Flexible process runtime with smaller change impact. Processes change over time and the runtime architecture should support these process changes with minimal cost. The unawareness of interaction partners in an event-based system enables flexible process change. By means of independently resubscribing to specific events, process fragments change their event rule, affecting and changing the global process flow. New process fragments can be added to the runtime architecture by simply letting them subscribe to existing events, after which they take part in the process execution collaboration.

Due to this high degree of decoupling, re-specifying and redeploying a previously deployed process model will have a reduced impact on the already running components compared to using a request style of distributed execution. Request-based distributed execution is a distribution approach where *directed* execution requests are sent from one process fragment to another, instead of an *indirect* collaboration of the process fragments through event communication. In request-based execution, the logic of the next step in the process flow is located at the sender (request logic), instead of at the receiver (event rule) (Mühl et al., 2006).

Table 1 quantifies the change impact according to the process change patterns introduced by Weber et al. (Weber et al., 2008). The change impact of using an event-based communication style in distributed process execution is compared with other non event-based (non content-routing) process distribution techniques, i.e. request based distribution like is done in (Chafle et al., 2004). The change impact for a certain process change pattern is defined as:

Definition. *Change impact* $= \sum_{i=0}^{n} w_i(c(f_i))$

with w_i *a weight indicating the importance of a specific process fragment;* f_i *a process fragment; and*

$c(f_i) = \begin{cases} 1 \; if \; the \; event \; rule \; or \; request \; operation \; changed \\ 0 \; if \; nothing \; changed \end{cases}$

For a specific pattern, the number of process fragments that need to be changed is counted, where a similar weight for each fragment is assumed. For example, inserting a new task between the sequential tasks *Create Bill* and *Send Bill* (change patten AP1-Serial Insert), has a change impact of 2 for event-based execution: a new component that needs to be inserted in the architecture, and an event rule change for the *succeeding* component. When using a request-style of communication, also 2 components need to change: the new inserted component and the

Table 1: Change impact when changing the global process flow

Change Pattern	Event Orchestration	Request Orchestration
AP1-Serial Insert	2	2
AP1-Parallel Insert	2	3
AP1-Conditional Insert	2	3
AP2-Delete Process Fragment	2	2
AP3-Serial Move	3	3
AP3-Parallel Move	3	4
AP3-Conditional Move	3	4
AP4-Replace Process Fragment	2	2
AP5-Swap Process Fragment	3	3
AP8-Embed Process Frag. in Loop	2	2
AP9-Parallelize Process Frag.	n + 1	n + 2
AP10-Embed Process Frag. in Conditional Branch	2	2
AP11-Add Control Dependency	1	2
AP12-Remove Control Dep.	1	2
AP13-Update Condition	1	1
AP14-Copy Process Fragment	2	3
Total Components to Change	32	40

With n = |elements in a process fragment|

(patterns AP6-7 were left out, due to not relevant)

component *preceding* the new component in the flow (e.g. by changing the *invoke* and *partnerLink* elements in a BPEL process).

From table 1 it can be seen that, compared to other distributed approaches, in 8 out of 16 cases, changing the process flow with event-based execution has lesser impact on the already running infrastructure and has a similar impact for the other cases. This is a substantial benefit, because change can be costly, certainly if the components are highly distributed and not readily available for change (e.g. other people are responsible).

With proper tool support (change detection) and process instance management (van der Aalst and Jablonski, 2000), only a limited amount of components need to be redeployed, while the rest can be left untouched (and running) in the architecture.

Note that the previous comparison compared the change impact in a distributed process execution approach, in which either an event-based or request-based distribution is used. When comparing the change impact of the event-based distributed process execution with the change impact of its centralized counterpart, there is a higher change impact for the distributed approach. Indeed, extra coordination overhead is introduced by distributing the process logic (see also Section 6 and Section 7.5). In Table 1, a centralized execution would always have a change impact of 1.

Flexible Process Deployment. Loose coupling and content-based routing of publish/subscribe messages enables process fragments to be deployed anywhere in the IT-architecture. This allows for the distribution of the process fragments, either because of managerial reasons, like security, or because of technical reasons, like performance. Process fragments can for example be deployed in an architectural scheme which enhances concurrent execution or minimizes network traffic (Chafle

et al., 2004). See also section 4.1.

Second, process deployments are able to change over time (due to organizational change, failure, overload, ...). A process fragment can for example be relocated from one organizational entity to another, by either physically moving the process fragment's engine and publish/subscribe wrapper (possible because of the location independent content-based routing); or by replacing the to-be-changed entity with a duplicated entity on the new location. The old entity unsubscribes to all events, while the new entity subscribes to the same event rule.

Third, the autonomy of process fragments (because of the added starting logic) facilitates the distribution of responsibility and control of the process fragments. The fragment's owner is not only responsible for its content, but also for the starting logic of the process fragment. This enables *full* control over the process fragment and gives the owner the power of changing the content as well as the starting logic of the process fragment.

Plugin Ability. Because of the unawareness of interaction partners in an event-based (publish/subscribe) communication style, entities can freely enter or leave the communication architecture. This means that extra features like process monitoring, dashboards and management tools are easily added into the execution architecture, without changing the actual execution entities (process engines). For example, when a monitoring tool subscribes to all events in the publish/subscribe architecture, it is ready to receive every event notification and enables the non-intrusive monitoring of the process execution. Event notifications are also a vital part of the process execution, so that observed event notifications *always* relate to an executed part of the process. Note that the plugin ability *should* be an advantage of any BPM system. The use of the events and the publish/subscribe architecture only simplifies this ability.

7.3. Performance Evaluation

Decentralization adds coordination overhead to the process execution. Network (event) messages have to be routed from one fragment to another. Moreover, each fragment has to evaluate its event rule each time an event message arrives. In contrast, in the centralized approach a simple state marker provides all the necessary information to coordinate the process flow.

To test any potential overhead in the distributed approach, we performed a couple of performance tests in which we measured the throughput of process execution (process instances completed per minute) as a function of request rate and as a function of the data transmitted to the process fragments. We compare the performance of the distributed event-based approach with the centralized approach (one process engine) and with a centralized replicated approach. To optimize the replicated approach, a load balancer is used in combination with multiple replicated engines (2 and 4). The load balancer uniformly distributes all incoming instantiation requests across the replicated engines. Note that we do not compare our event-based decentralization with other process decentralization techniques (see section 8), because the performance in the distributed approach is highly dependent on the distribution grouping chosen (see section 4.1). Since we do not choose a specific distribution grouping and focus on the provision of an enactment environment

in which the distribution can be defined arbitrarily, and since we want to avoid a positive bias in favor of the distributed approach, we look at the *worst case* distribution in which every task-level fragment is distributed in the environment and compare this with (the optimized) centralized process execution.

All tests were done on a LAN network with 8 PCs, each with 4GB of memory and an Intel Core 2; 3,17GHz processor. For the centralized execution setup, one PC was equipped with a process engine (Alfresco, 2012) which coordinates the entire process model, one PC served as the client sending process creation requests and one machine ran a simple web service which returned a response to each invocation. In the replicated setup, the single process engine is replaced by multiple PCs and each client request is uniformly distributed across these process engines. In the decentralized setup, the replicated engines are replaced by engines only coordinating a small fragment of the process flow. Each fragment is executed on a different PC. One additional machine is used to run the Siena publish/subscribe service. As test model, a similar model to the one used in (Li et al., 2010) is used. It contains 5 tasks, with 2 tasks in parallel and 2 tasks in XOR-relation. Each task invokes the simple web service. As a default, the response message size to and from the web service is 512 bytes.

Figure 15 shows the throughput measured with varying process request rate. Each configuration ran 5 minutes, and the average throughput is taken to represent the data point on the graph. From the figure it is apparent that a significant difference is measured between the centralized and decentralized execution from a 1000 requests per minute and up ($F = 6.78$; $p = 0.009$). The performance decline in process execution can be attributed to the processing of XML documents for either data handling or service invocations. Extraction of data from xml-containers for e.g. variable assignments in the process model can involve complicated operations (using for example XPath-queries). Also, the creating, marshalling and unmarshalling of SOAP messages requires significant processing power (Chafle et al., 2004; Kohlhoff and Steele, 2003). To evaluate the stress of handling SOAP messages on a process engine, Figure 16 shows a micro benchmark on the CPU usage of a simple request-reply to/from a web service. At 600 rpm, invoking a web service already uses an average of 9.5% of CPU time. By increasing the rpm to 1500, the processor usage significantly increases to an average of 12.5% ($F = 87.8$; $p < 0.00001$). Handling multiple process instances, with each multiple parallel tasks (service invocations) at a high request rate, saturates the CPU, thus decreasing the throughput of the process execution. Since the distributed approach can leverage the execution power of more resources, performance increases compared to centralized execution.

A similar reasoning can be followed for the replicated setup. The replicated setup does not suffer from the performance decline, as more computing power can be used. When comparing the decentralized setup with the replicated setup, no significant difference is found. The overhead cost of decentralization stays limited. Fragments send event messages in a fire-and-forget manner, hereby never directly interacting with other fragments. Event notifications are also only routed to fragments which are interested in those notifications, i.e. notifications are only routed to fragments which are directly sequence dependent. Moreover, the event messages are small and evaluation of an (LTL) event rule can be done by checking a simple state machine (Gastin and Oddoux, 2001). Only at very high request rates a small difference can be found. The overhead of creating reliable (tcp) network connections degrades the performance slightly. Note that the worst case distribution is used for the experiment. Employing a better distribution

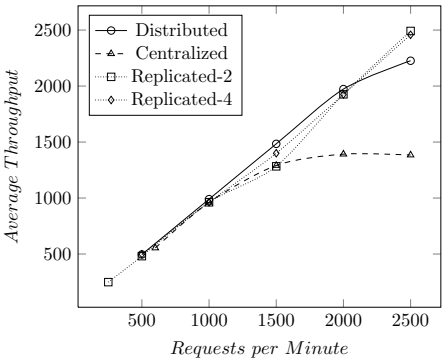


Figure 15: Average throughput with varying request rate

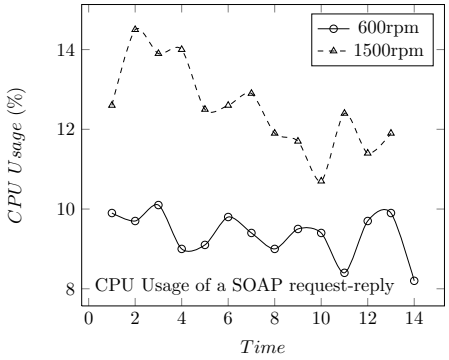


Figure 16: CPU usage of a SOAP invoke-reply operation

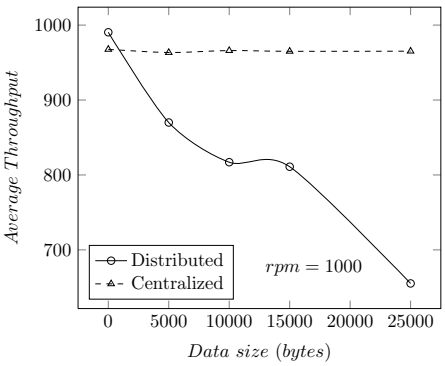


Figure 17: Average throughput with varying data size at 1000rpm

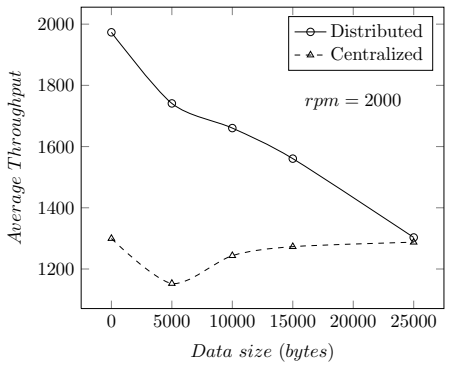


Figure 18: Average throughput with varying data size at 2000rpm

strategy (Chafle et al., 2004) will decrease the overhead.

A higher coordination overhead is however measured when transmitting larger data packets. Figure 17 and 18 show the average throughput with varying data size transmitted to every process fragment. For the distributed approach, every event notification is appended with the received data, routing this data in its entirety to other process fragments (first option of data handling in Section 5.3). Both figures show a decrease in throughput with an increase of data size for the distributed approach. The drop in throughput is due to the delays in network traffic involved when sending data over the network. The event service itself does not require more time to process the event notification, since only the header of the notification, containing the `event-id`, is inspected. Data transmission through event notifications is only limited by the bandwidth of the network. Each time an event notification is created, the data needs to be routed through the network again (and multiple times for duplicated notification messages). This restriction does not apply for centralized execution (independent of the use of replication), since the data element is only sent and received at the start of the process flow. Through-

put for centralized execution thus stays constant over varying data sizes. Although the centralized approach outperforms the distributed approach when large amounts of data come into play, it is very unlikely that every node in the process model needs the entirety of the data received at the start of the process. To reduce the networking delays and hereby increasing process throughput, other approaches can be used to handle data, instead of transmitting and duplicating it with every event notification (see Section 5.3). For example, when a central data store is used no data is transmitted along with the control flow event notifications, hereby creating a process execution performance similar to Figure 15.

In conclusion, the coordination overhead in the decentralized setup stays limited and even outperforms the centralized setup (one engine) when the process instantiation rate is high. Moreover, the decentralized approach performs similar to a replicated setup (both can leverage more computing power) and the coordination cost of the decentralized approach stays limited. The coordination overhead is however present when relatively large amounts of data are transmitted along with the event notifications. Hence, a decrease in bandwidth will have a negative impact on the performance in the decentralized setup (which is no issue for the replicated approach).

7.4. Evaluation of Robustness

To quantify the fault-tolerance of the distributed execution (REQ3), we measured the throughput of process execution during failure of some system components (fragments) and compared this to the centralized approach (with and without a load-balancer). When no load balancer is used in the centralized approach, the entire process control stalls (the actual services or human performers are not triggered anymore) if the (single) process engine fails, i.e. *all* process instances are interrupted. If a load balancer is used, the process instances managed by the failed engine are interrupted (no matter the state of the process instance), while all others can continue their execution. In the distributed environment, the failure of a specific fragment will interrupt the execution of process instances which reach a state where the failed engine should handle the further execution. Process instances in a state before or after the failed fragment can continue their execution.

Figure 19 shows the results of the measurements performed to test the fault-tolerance. An experiment is setup where a client sends process instantiation requests at a constant rate (500rpm). At time 4, we stop the execution of a specific component in the enactment environment to simulate a failure.

For the centralized approach, we distinguish between 4 scenarios:

1. A single process engine, coordinating every process instance and handling every process instantiation request;
2. A load balancer with 2 replicated process engines;
3. A load balancer with 5 replicated process engines;
4. A load balancer with 7 replicated process engines (equal to the amount of fragments);

When a load balancer is used[11], each engine only handles a part of the total number of process instances. Once a process instance is assigned to a specific engine, it will

[11]Note that a very simple load balancer is used. The load balancer uniformly distributes the process instantiation requests across all available process engines.

be coordinated in its entirety by that specific engine (in contrast to the fragmented approach). A failure during these scenarios will only affect *one* of the available process engines. In the first case this therefore means that all process instances are interrupted. In the second case, this means that a total of #`process-instances`/2 are interrupted, etc.

In the distributed approach we distinguish between 3 scenarios:

1. The component that is stopped (fails) is an engine running a *blocking* process model fragment. With a blocking fragment we mean a process part that is contained in *all* execution paths of the process model and which is *not run in parallel* with another activity in the process model. For example: the 'Deliver Pizza' task in figure 2.
2. The component that is stopped is an engine running a *parallel* process model fragment. With a parallel fragment we mean a process part that is contained in *all* execution paths of the process and which is *always run in parallel* with another activity in the process model. For example: the 'Package Order' task in figure 2.
3. The component that is stopped is an engine running a *choice* process model fragment. With a choice fragment we mean a process part that is only contained in *some* execution paths of the process. For example: the 'Create Side Dish' task in figure 2.

After 5 time intervals, the failed components are restarted. The results are presented in figure 19.

When only one process engine is used, as soon as the engine is stopped the throughput drops to zero. No single process instance is able to complete when the engine that controls the entire process flow has failed. Only when the engine is restarted (time 9) the process execution is resumed. Note that the client kept sending process instantiation requests, even during downtime. This means that a backlog has to be handled when the execution is resumed. This backlog is indicated by the data peak at time 23. When a *blocking* or a *parallel* fragment is stopped, a similar throughput is measured. The downtime is however smaller than in the centralized approach. This is due to the fact that process instances that reside in a state behind the failed process fragment engine can still execute and finish correctly. Moreover, the backlog that needs to be handled is smaller and the effect of the failure is only noticeable at a later time. When a *choice* fragment is stopped, no downtime is measured. There are still process instances that are not affected by the failed fragment, which still execute correctly.

When a replicated process engine is used, throughput during downtime increases when the number of replicated engines increases. Indeed, with more engines, a smaller number of instances has to be handled by each engine. If one engine fails, less process instances are affected by the failure. Moreover, the backlog that is build up during downtime of one of the replicated engines could be decreased even further if a better load balancer is used[12].

There is a clear distinction between the robustness of using a single process engine and using multiple engines (either replicated or fragmented). The difference between the fragmented environment and the replicated environment is however less clear. In the replicated environment, there are *always some* process instances affected by the

[12] A better load balancer would stop sending instantiation requests to the interrupted process engine.

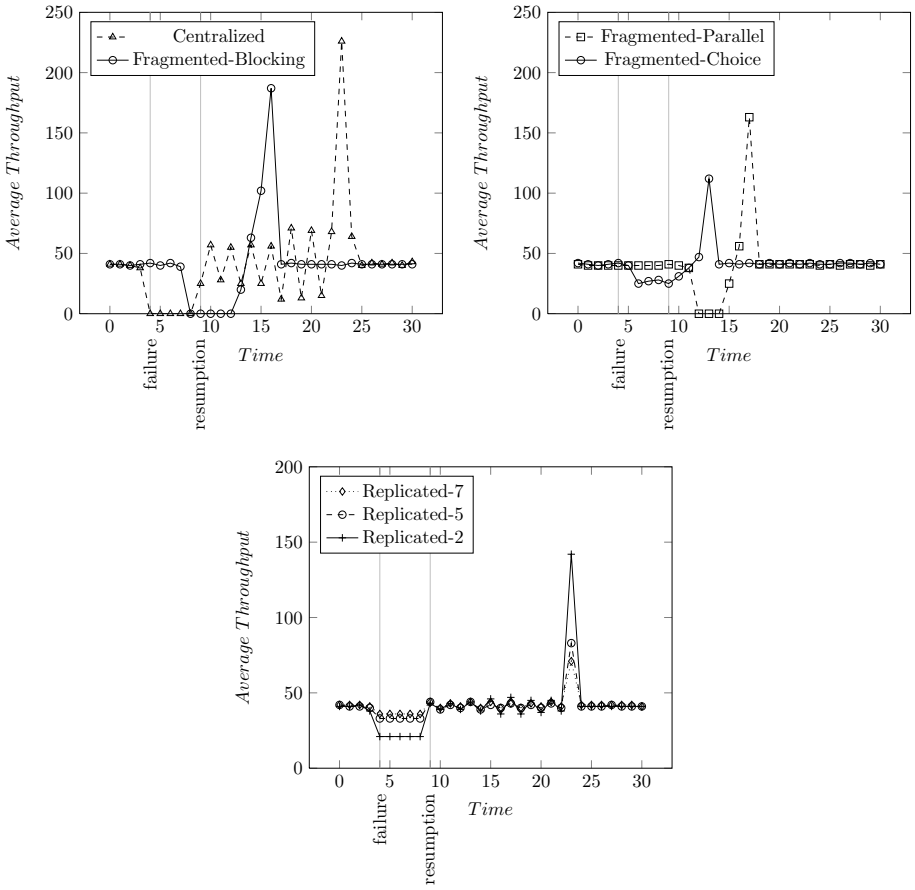


Figure 19: Fault-tolerance of the execution architecture

interruption: the instances handled by the failed process engine. In the fragmented environment, *all* instances are *potentially* affected by the interruption. It is possible that each running instance is already past the failed fragment, whereby no single instance is affected by the failure. On the other hand, when all instances are in a state before the failed fragment, all these instances will eventually get interrupted (when they reach the blocking, failed fragment). In summary:

- With N replicated engines and a uniform distribution of the process instances: a total of #`process-instances`/N are *always* impacted by the failure.
- With N fragmented engines: *all* instances are *potentially* impacted by the failure.

When resilience to failure is concerned, the fragmented approach is not more (or less) robust than the replicated execution environment. It is situationally dependent. Note however that the event-based distributed environment adds another layer of infrastructure: the publish/subscribe architecture. The event brokers can also fail, hereby interrupting the fragmented process execution. Some solutions do however exist that build a robust publish/subscribe architecture by building redundant notification routes (Chand and Felber, 2004), replicating event brokers (Bhola et al., 2002), replicating subscriptions (Baldoni et al., 2004) or clustering brokers (Jafarpour et al., 2008).

7.5. Limitations

Besides the performance increase and the advantages of flexible deployment and runtime, there are however still limitations to the approach. There is an obvious overhead when decentralizing processes that are not executed very frequently. Multiple computers are necessary to achieve the fully distributed architecture together with a deployment of the event architecture in the company's IT-infrastructure (instead of using a simple process engine). The performance evaluation shows that the advantages manifest itself above 1000 requests per minute. For business processes which never receive that many process requests, distributing the flow would be overkill. The same can be concluded if there is no need in distributing process control or visibility.

A second disadvantage is the increased network traffic. Event notifications will populate the network when distributing the process flow. Using simple event messages does not significantly increase the bandwidth usage, as the event message only contains 2 simple ids. When, however, large amounts of process data (e.g. customer info, product info, ...) are included in the notification broadcasts, bandwidth can become saturated. To overcome this, other solutions have to be considered to route data to the correct process fragment(s) (see Section 5.3).

A third limitation are the limited possibilities for easy coordination between different process fragments, which has an impact on the process elements that can be transformed into a distributed execution. Cancellation regions and condition-places need additional event communication to be implemented. The OR-gateway semantic requires full process state knowledge and synchronization with other active branches in the process. This is cumbersome in the distributed environment and contradicts the idea of autonomous, loosely coupled process fragments. To overcome this problem, some or-gateway constructs can be redesigned into a combination of XOR- and AND-gateways (as is done for the pizza company in Figure 2), which can then still be transformed to a decentralized

execution. Nonetheless, the most used process constructs are covered in the given transformation (zur Muehlen and Recker, 2008) and enable the process to be executed in a distributed event-based manner.

The increased coordination cost also increases the change overhead in the distributed system compared to a centralized execution approach. As already mentioned in Section 6, processes change over time and the execution architecture should provide support for these process changes. In a centralized approach, a new process model can simply be deployed onto the process engine, which provides a single point where all necessary information is available for process change and evolution (Rinderle et al., 2004). In the distributed approach however, extra coordination overhead is introduced because the state of the running process instances is scattered in the environment. Moreover, an extra system has to be introduced to correctly discover the impact of a process model change on individual process fragments (see Section 6).

Another limitation is the complexity of the generated event rules. We presented an automated approach that does not require the intervention of the process modeler: a process model is provided, which is automatically split and distributed. However, if a manual intervention is needed, the rules have to be understood. For this purpose, a visualization can be used which is able to provide for a comprehensible representation of the LTL rules (Brambilla et al., 2005).

8. Related Work

In the domain of PAIS, the problem of centralized process execution is recognized by many researchers and the opportunities of workflow fragmentation have been studied accordingly. The main goal of the workflow fragmentation research is to increase the scalability of the process enactment (Chafle et al., 2004) and the separation of control over different process parts (Khalaf and Leymann, 2006).

Wodtke et al. (1996) present the MENTOR project. The project describes a distributed enactment system, that starts from a process defined in state and activity charts. After specification, activities are assigned to each state in the state graph and a partitioning is performed. The outcome of the partitioning is a set of orthogonal components, one for each activity. These partitioned state charts can hereafter be re-grouped into a *distribution grouping*, where each partitioned subset of the state chart can be executed by a different engine. The Self-Serv architecture for web services composition (Benatallah et al., 2003) also relies on state charts for process modeling (service composition). The execution of the composite service relies on a peer-to-peer coordination scheme. The peer-to-peer coordination is achieved by generating a *state coordinator* for each state in the composite service's description. The state coordinator receives messages from other state coordinators, checks if preconditions are met and invokes, if possible, an included service. As postcondition, the state coordinator notifies any dependent coordinators of its completion, making it responsible for *directly* invoking its own successors. The biggest difference with our approach is that we start from a process model describing the sequence dependencies between activities and not from state charts.

Bauer and Dadam (1997) present a distribution approach for the ADEPT workflow management system. Workflow servers are assigned to different partitions of the original process model. Control over the workflow is handed over (migrated) from server to server. To this end, a description of the state of a workflow instance is transmitted

from the current server to a target server before it can take over control. An additional goal presented in the $\text{ADEPT}_{\text{distribution}}$ technique is the minimization of communication overhead (Bauer and Dadam, 2000). A cost model is introduced that allows flexible and dynamic server assignments, hereby limiting the communication overhead. $\text{ADEPT}_{\text{distribution}}$ does however assume that the *full* process model is duplicated across every distributed execution node. In contrast, the distribution technique of Fdhila et al. (2009) does create individual process fragments out of the original process model. A process flow is decentralized using dependency tables created from the original process model. The dependency tables are used to create subprocesses, where *fictive* (zero execution time) activities are added inside fragments to serve as synchronization elements. Interconnection between the subprocesses is achieved by sending *direct* messages from one fragment to another (received at a specific point inside the fragment). The technique is extended in (Fdhila and Godart, 2009) to support more advanced process model constructs in the decentralization.

Chafle et al. (2004) propose an approach to decentralize a BPEL process into several subprocesses, where each subprocess is deployed and executed by a different coordinator. The work is primarily focused on optimizing the fragmentation of the original process model so as to reduce the eventual network traffic in distributed execution. Heuristics are used to improve the fragmentation. Any resulting fragments synchronize their execution at enactment by directly invoking any dependent fragments. Khalaf and Leymann (2006) split a BPEL process flow according to predefined swim-lanes. The split is performed according to the data flow between the different fragments. The decentralization is based on an extended metamodel of BPEL that explicitly defines data dependencies and results in BPEL-compliant process fragments. Only for loops and fault/compensation handlers extra middleware is required. Each resulting fragment holds the responsibility of directly invoking any dependent fragment (using invoke-receive links in BPEL). Another similar distributed architecture is presented by Zhai et al. (2007). A BPEL process is split according to the data flow in the process description. Additionally they present an optimized fragmentation technique by building a Parallel Flow Graph and optimizing the graph based on data flow analysis.

The METEOR2 (Sheth et al., 1996) system proposes a fragmentation and distributed workflow execution based on rules. Each task in the predefined workflow is assigned to a task manager. The task manager is subdivided into pre-activation, task activation and post-activation. The pre-activation of the task manager contains a rule of messages that have to be received before task activiation can start. In the post-activation part of the task manager, other depending (succeeding) task managers are activated. The task manager itself is therefore responsible of triggering dependent process fragments. A similar idea is presented by Casati et al. (1996), who describe a precise operational semantic to workflow enactment through active rules. While the presented technique is created for the incorporation of active databases in the operational model of workflow enactment and not for distribution purposes, it could be used to run a fragmented process model. Active rules (Event-Condition-Action, ECA) are defined that state, according to database insertion, deletion and update events, when other (database) actions have to happen. Specific database actions are related to workflow actions (creation of a task instance, process instance, state change, etc.). The use of active rules in a distributed environment is also presented by Ceri et al. (1997). Their distribution approach does however not describe the distribution of (logical) process fragments, but rather focuses

on a client/server database architecture and a distributed object model. In complex event processing (CEP) (Luckham, 2002), event rules and event stream interpretations are used to build monitoring and reactive applications. CEP is complementary to our approach, in the sense that the already existing CEP engines can be used to evaluate the event rules, hereby already capturing the coordination logic at the event bus. For this purpose the LTL rules can be rewritten in a rule language suitable for CEP.

Li et al. (2010) decentralize BPEL processes and use an event-based communication mechanism to synchronize the BPEL fragments (instead of the invoke-receive links). It is argued that the event-based communication achieves a much more flexible execution environment, which allows a more precise control over load balancing and replication needs and allows a non-intrusive fine grained monitoring service. Different than our approach, each original process element (including gateways) becomes a small fragment on its own and generates event messages (and hereby network traffic). The decentralization can also not cope with more advanced BPEL process constructs (like Khalaf and Leymann, 2006). The runtime event-based architecture does however provide some significant advantages. The latter is also recognized by Fjellheim et al. (2007) who describe a framework for flexible process modeling whereby a process, defined in an UML Activity Diagram, is translated into an event based application. While created for flexibility, the architecture could be used to run distributed process model fragments. Each task of the original process model is translated to an autonomous entity that listens to necessary event notifications. Using an event-based interconnection scheme creates a highly flexible environment. The support for workflow patterns is however limited. Only very basic gateways are supported by the architecture and the UML Activity Diagrams.

In Brambilla et al. (2006), a method to model process- and service- enabled web applications is proposed. The authors also mention different methods for distributed process coordination. The distributed coordination is based on a given process model in which each distributed process part is modeled in a different pool. Two coordination classifications are proposed: nested coordination and generalized coordination. Our approach fits in the latter class, as no restrictions are put on the possible distribution grouping (see section 4.1). The same authors also propose a BPMN to LTL mapping which is used in compliance checking of web applications (Brambilla et al., 2005). This mapping is used to visualize and simplify the construction of LTL statements. The approach can be used to lower the complexity of our generated event rules (see the discussion in section 7.5). In Desai et al. (2009) the Amoeba approach is presented which describes business protocols to explicitly model cross-organizational processes. Furthermore, techniques for evolution of requirements and change propagation are also presented for these business protocols. The approach differs from our technique, where we do not require the explicit modeling of the fragment interactions. We assume that a standard end-to-end process is modeled and this model needs to be distributed, preferably without any intervention by the original process modeler (see also figure 1).

The technique of rule checking as a restriction or triggering mechanism of processes is also used by declarative process modeling languages and execution models (van der Aalst et al., 2009; Sadiq et al., 2005; Bhattacharya et al., 2009). In a sense, the declarative process model research is compatible with our approach in that the process engine mechanics to run the distributed process fragments are similar to the enactment engine interpreting the declarative LTL rules (where the LTL-Rules are the restrictions on the process fragments). The main difference is that we start from an imperative process

model and transform this to a distributed model with declarative rules (REQ 1), while in declarative modeling, models are defined in a declarative way right from the start.

Our approach leverages the flexibility of the event-based communication architecture for business process execution like (Li et al., 2010) and (Fjellheim et al., 2007), supports advanced workflow concepts like (Fdhila and Godart, 2009) by providing an automatic derivation of the rules like (Casati et al., 1996) and uses a fragmentation and distribution approach like (Wodtke et al., 1996) (allowing an arbitrary distribution). As a result, the approach proposed in this paper integrates the positive results achieved by different researchers into a single execution architecture. Moreover, we discuss the flexibility advantages in the context of process model changes and provide an indication of the robustness of the architecture. In Hens et al. (2013) we extend the architecture and show how the flexibility advantage can be used to quickly perform change adaptations to the deployed process model.

9. Conclusion and Future Research

In this paper we investigated the use, advantages and disadvantages of an event architecture in distributed process execution. First, we have shown how, after process modeling and distribution definition, the modeled process flow can be automatically transformed into (logically-) different process model fragments. Each process model fragment is complemented with an event rule, stating its starting logic and hereby creating an autonomous, self-serving process model component. Next, we described the deployment and execution of the process fragments. Each fragment is deployed on a dedicated process engine, wrapped in a publish/subscribe wrapper which is able to communicate with the event architecture. Subscriptions, publications and notifications of event messages create a collaboration between the different fragments, reconstructing the global process execution. Third, the execution architecture is evaluated by formally validating the transformations involved and comparing the performance of distributed execution with centralized process execution. The coordination cost of the distributed approach stays limited and behaves similarly to a replicated centralized approach. When compared to request based distribution, using an event based communication scheme also creates a loosely coupled process execution environment, where process management and monitoring tools can be easily added into the execution environment. No structural changes are necessary to the process fragments, since they already publish event notifications, indicating their successful completion.

Future research involves the inclusion of non-local workflow patterns in the fragmentation algorithm, like the OR-Join. A possible way to include the OR-join semantic in the fragmentation and distribution is to make use of *negative events*. To use of negative events enables the possibility to adapt *death-path elimination* techniques and therefore the OR-Join semantic in distributed event-based process execution.

References

Alfresco, 2012. Activiti bpm platform. http://www.activiti.org/.

Baldoni, R., Beraldi, R., Querzoni, L., Virgillito, A., 2004. A self-organizing crash-resilient topology management system for content-based publish/subscribe. In: Third International Workshop on Distributed Event-Based Systems. Citeseer, p. 3.

Basten, T., 1996. Branching Bisimilarity is an Equivalence indeed! Information Processing Letters 58, 141–147.

Bauer, T., Dadam, P., 1997. A distributed execution environment for large-scale workflow management systems with subnets and server migration. In: International Conference on Cooperative Information Systems. IEEE, pp. 99–108.

Bauer, T., Dadam, P., 2000. Efficient distributed workflow management based on variable server assignments. In: Advanced Information Systems Engineering. Springer, pp. 94–109.

Benatallah, B., Sheng, Q., Dumas, M., 2003. The self-serv environment for web services composition. Internet Computing, IEEE 7 (1), 40–48.

Bhattacharya, K., Hull, R., Su, J., et al., 2009. A data-centric design methodology for business processes. Handbook of Research on Business Process Modeling, 503–531.

Bhola, S., Strom, R., Bagchi, S., Zhao, Y., Auerbach, J., 2002. Exactly-once delivery in a content-based publish-subscribe system. In: Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on. IEEE, pp. 7–16.

Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I., 2006. Process modeling in web applications. ACM Transactions on Software Engineering and Methodology (TOSEM) 15 (4), 360–409.

Brambilla, M., Deutsch, A., Sui, L., Vianu, V., 2005. The role of visual tools in a web application design and verification framework: a visual notation for ltl formulae. In: Proceedings of the 5th international conference on Web Engineering. Springer-Verlag, pp. 557–568.

Carzaniga, A., Rosenblum, D. S., Wolf, A., 2001. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems 19, 332–383.

Casati, F., Ceri, S., Pernici, B., Pozzi, G., 1996. Deriving active rules for workflow enactment. In: Database and Expert Systems Applications. Springer, pp. 94–115.

Ceri, S., Grefen, P., Sanchez, G., 1997. Wide-a distributed architecture for workflow management. In: Research Issues in Data Engineering, 1997. Proceedings. Seventh International Workshop on. IEEE, pp. 76–79.

Chafle, G., Chandra, S., Mann, V., Nanda, M., 2004. Decentralized orchestration of composite web services. In: Proceedings of the 13th international World Wide Web conference. ACM Publications, pp. 134–143.

Chand, R., Felber, P., 2004. Xnet: a reliable content-based publish/subscribe system. In: Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on. IEEE, pp. 264–273.

Chapell, D., 2004. Enterprise Service Bus: Theory in Practice. O'Reilly Media.

Decker, G., Dijkman, R., Dumas, M., Garca-Bauelos, L., 2008. Transforming bpmn diagrams into yawl nets. In: Dumas, M., Reichert, M., Shan, M.-C. (Eds.), Business Process Management. Vol. 5240 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 386–389.

Desai, N., Chopra, A. K., Singh, M. P., 2009. Amoeba: A methodology for modeling and evolving cross-organizational business processes. ACM Transactions on Software Engineering and Methodology (TOSEM) 19 (2), 6.

Dumas, M., van Der Aalst, W., Ter Hofstede, A., 2005. Process-aware information systems. Wiley.

Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A., 2003. The many faces of publish/subscribe. ACM Computing Surveys 35, 114–131.

Faraboschi, P., Fisher, J., Young, C., 2001. Instruction scheduling for instruction level parallel processors. Proceedings of the IEEE 89 (11), 1638–1659.

Fdhila, W., Godart, C., 2009. Toward synchronization between decentralized orchestrations of composite web services. In: Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on. IEEE, pp. 1–10.

Fdhila, W., Yildiz, U., Godart, C., 2009. A flexible approach for automatic process decentralization using dependency tables. In: Proceedings of the 2009 IEEE International Conference on Web Services. IEEE Computer Society, pp. 847–855.

Fjellheim, T., Milliner, S., Dumas, M., Vayssiere, J., 2007. A process-based methodology for designing event-based mobile composite applications. Data & Knowledge Engineering 61, 6–22.

Gastin, P., Oddoux, D., 2001. Fast ltl to büchi automata translation. In: Computer Aided Verification. Springer, pp. 53–65.

Gray, J., Siewiorek, D., sept. 1991. High-availability computer systems. Computer 24 (9), 39–48.

Hallerbach, A., Bauer, T., Reichert, M., 2010. Configuration and management of process variants. Handbook on Business Process Management 1 (1), 237–255.

Hens, P., Snoeck, M., De Backer, M., Poels, G., 2011. A petri net formalization of a publish-subscribe process system. In: FBE Research Report KBI1114, Leuven.

Hens, P., Snoeck, M., De Backer, M., Poels, G., 2013. Process evolution in a distributed process execution environment. International Journal on Information System Modeling and Design 4 (2), 65-90.
Hens, P., Snoeck, M., De Backer, M., Poels, G., 2014. Verification of change in a fragmented event-based process coordination environment. Accepted for IEEE Transactions on Services Computing.
Hollingsworth, D., 1995. The workflow reference model. WfMC Documents, 1995.
Jafarpour, H., Mehrotra, S., Venkatasubramanian, N., 2008. A fast and robust content-based publish/subscribe architecture. In: Network Computing and Applications, 2008. NCA'08. Seventh IEEE International Symposium on. IEEE, pp. 52–59.
Jennings, N., Norman, T., Faratin, P., OBrien, P., Odgers, B., 2000. Autonomous agents for business process management. Applied Artificial Intelligence 14 (2), 145–189.
Khalaf, R., Kopp, O., Leymann, F., 2008. Maintaining data dependencies across bpel process fragments. International Journal of Cooperative Information Systems 17, 259–282.
Khalaf, R., Leymann, F., 2006. Role-based decomposition of business processes using bpel. In: Proceedings of the International Conference on Web Services. IEEE Computer Society, pp. 770–780.
Kohlhoff, C., Steele, R., 2003. Evaluating soap for high performance business applications: Real-time trading systems. In: Proceedings of WWW2003. IW3C2, pp. 262–270.
Kong, J., Jung, J., Park, J., 2009. Event-driven service coordination for business process integration in ubiquitous enterprises. Computers & Industrial Engineering 57, 14 – 26.
Li, G., Muthusamy, V., Jacobsen, H., 2010. A distributed service-oriented architecture for business process execution. ACM Transactions on the Web 4, 2:1–2:33.
Luckham, D. C., 2002. The power of events. Vol. 204. Addison-Wesley Reading.
Mendling, J., Moser, M., Neumann, G., 2006. Transformation of yepc business process models to yawl. In: Proceedings of the 2006 ACM symposium on Applied computing. ACM Publications, pp. 1262–1266.
Monsieur, G., Snoeck, M., Lemahieu, W., 2012. Managing data dependencies in service compositions. Journal of Systems and Software.
Mühl, G., Fiege, L., Pietzuch, P., 2006. Distributed Event-Based Systems. Springer-Verlag, New York.
Object Management Group, June 2010. Bpmn 2.0. http://www.omg.org/cgi-bin/doc?dtc/10-06-04.
Pnueli, A., 1981. The temporal semantics of concurrent programs. Theoretical Computer Science 13, 45 – 60.
Recker, J., 2010. Opportunities and constraints: the current struggle with BPMN. Business Process Management Journal 16, 181–201.
Reichert, M., Rinderle, S., Dadam, P., 2003. On the common support of workflow type and instance changes under correctness constraints. In: Cooperative Information Systems. pp. 407–425.
Rinderle, S., Reichert, M., Dadam, P., 2004. Correctness criteria for dynamic changes in workflow systems: a survey. Data & Knowledge Engineering 50 (1), 9–34.
Sadiq, S. W., Orlowska, M. E., Sadiq, W., 2005. Specification and validation of process constraints for flexible workflows. Information Systems 30 (5), 349–378.
Schneider, F., 1982. Synchronization in distributed programs. ACM Transactions on Programming Languages and Systems 4, 125–148.
Sheth, A., Kochut, K., Miller, J., Worah, D., Das, S., Lin, C., Palaniswami, D., Lynch, J., Shevchenko, I., 1996. Supporting state-wide immunization tracking using multi-paradigm workflow technology. In: Proceedings of the international conference on very lare databases. Citeseer, pp. 263–273.
Sistla, A., Clarke, E., 1985. The complexity of propositional linear temporal logics. Journal of the Association for Computing Machinery 32, 733–749.
Tian, X., Chen, Y., Girkar, M., Ge, S., Lienhart, R., Shah, S., 2003. Exploring the use of hyper-threading technology for multimedia applications with intel® openmp compiler. In: Parallel and Distributed Processing Symposium, 2003. Proceedings. International. IEEE, pp. 8–pp.
van der Aalst, W., 1998. The application of Petri nets to workflow management. Journal of Circuits Systems and Computers 8, 21–66.
van der Aalst, W., Jablonski, S., 2000. Dealing with work flow change: identification of issues and solutions. International Journal of Computer Systems Science and Engineering 15, 267–276.
van der Aalst, W., Pesic, M., Schonenberg, H., 2009. Declarative workflows: Balancing between flexibility and support. Computer Science-Research and Development 23, 99–113.
van der Aalst, W., Ter Hofstede, A., 2005. YAWL: yet another workflow language. Information Systems 30, 245–275.
van der Aalst, W., Ter Hofstede, A., Kiepuszewski, B., Barros, A., 2003. Workflow patterns. Distributed and parallel databases 14, 5–51.
van der Aalst, W. M. P., 2001. Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers 3 (3), 297–317.

van Dongen, B., de Medeiros, A., Verbeek, H., Weijters, A., van der Aalst, W., 2005. The prom framework: A new era in process mining tool support. In: Ciardo, G., Darondeau, P. (Eds.), Applications and Theory of Petri Nets. Vol. 3536 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1105–1116.

Weber, B., Reichert, M., Rinderle-Ma, S., 2008. Change patterns and change support features-enhancing flexibility in process-aware information systems. Data & knowledge engineering 66, 438–466.

Weber, B., Sadiq, S., Reichert, M., 2009. Beyond rigidity - dynamic process lifecycle support: A survey on dynamic changes in process-aware information systems. Computer Science - Research and Development 23 (2), 47–65.

Wodtke, D., Weissenfels, J., Weikum, G., Dittrich, A., 1996. The mentor project: Steps towards enterprise-wide workflow management. In: Conference on Data Engineering. pp. 556–565.

Ye, J., Song, W., 2010. Transformation of bpmn diagrams to yawl nets. Journal of Software 5, 396–404.

Zhai, Y., Su, H., Zhan, S., 2007. A data flow optimization based approach for bpel processes partition. In: IEEE International Conference on e-Business Engineering, 2007. ICEBE 2007. Ieee, pp. 410–413.

zur Muehlen, M., Recker, J., 2008. How much language is enough? theoretical and practical use of the business process modeling notation. In: Proceedings of the 20th international conference on Advanced Information Systems Engineering. Springer-Verlag, pp. 465–479.